{API}

Sails.js 官方文档（中英合辑）



译者 imfly
版本 0.1
时间 2015.10.24

Table of Contents

Sails.js Documentation (v0.11.x)

Guide and reference documentation for the 0.11.x release of Sails. Content for most sections on the Sails website (beta.sailsjs.org) is compiled from here.

Contributing to the docs

We welcome your help! Please send a pull request to **master** with corrections/additions and they'll be double-checked and merged as soon as possible.

Secondly, we are open to suggestions about the process we're using to manage our documentation, and to work with the community in general. Please post to the Google Group with your ideas- or if you're interested in helping directly, contact @fancydoilies, @ncrumrine, @loicsaintroch, or @mikermcneil on Twitter.

Assets

Overview

Assets refer to static files (js, css, images, etc) on your server that you want to make accessible to the outside world. In Sails, these files are placed in the `assets/` directory, where they are processed and synced to a hidden temporary directory (`.tmp/public/`) when you lift your app. The contents of this `.tmp/public` folder are what Sails actually serves - roughly equivalent to the "public" folder in express, or the "www" folder you might be familiar with from other web servers like Apache. This middle step allows Sails to prepare/pre-compile assets for use on the client - things like LESS, CoffeeScript, SASS, spritesheets, Jade templates, etc.

Static middleware

Behind the scenes, Sails uses the static middleware from Express to serve your assets. You can configure this middleware (e.g. cache settings) in `/config/http.js` .

`index.html`

Like most web servers, Sails honors the `index.html` convention. For instance, if you create `assets/foo.html` in a new Sails project, it will be accessible at `http://localhost:1337/foo.html` . But if you create `assets/foo/index.html` , it will be available at both `http://localhost:1337/foo/index.html` and `http://localhost:1337/foo` .

Precedence

It is important to note that the static middleware is installed **after** the Sails router. So if you define a custom route, but also have a file in your assets directory with a conflicting path, the custom route will intercept the request before it reaches the static middleware. For example, if you create `assets/index.html` , with no routes defined in your `config/routes.js` file, it will be served as your home page. But if you define a custom route, `'/': 'FooController.bar'` , that route will take precedence.

Default Tasks

Overview

The asset pipeline bundled in Sails is a set of Grunt tasks configured with conventional defaults designed to make your project more consistent and productive. The entire frontend asset workflow is completely customizable, while it provides some default tasks out of the box. Sails makes it easy to configure new tasks to fit your needs.

Here are a few things that the default Grunt configuration in Sails does to help you out:

- Automatic LESS compilation
- Automatic JST compilation
- Automatic Coffeescript compilation
- Optional automatic asset injection, minification, and concatenation
- Creation of a web ready public directory
- File watching and syncing
- Optimization of assets in production

Default Grunt Task Behavior.

Below are the Grunt tasks that are included in your Sails project as well as a small description of exactly what each does in your project. Also included are a link to the usage docs for each task.

clean

> This grunt task is configured to clean out the contents in the `.tmp/public/` of your sails project.
>
> usage docs

coffee

> Compiles coffeeScript files from `assets/js/` into Javascript and places them into `.tmp/public/js/` directory.
>
> usage docs

concat

> Concatenates javascript and css files, and saves concatenated files in `.tmp/public/concat/` directory.
>
> [usage docs]

copy

> **dev task config** Copies all directories and files, except coffeescript and less files, from the sails assets folder into the `.tmp/public/` directory.
>
> **build task config** Copies all directories and files from the .tmp/public directory into a www directory.
>
> [usage docs]

cssmin

> Minifies css files and places them into `.tmp/public/min/` directory.
>
> [usage docs]

jst

> Precompiles Underscore templates to a `.jst` file. (i.e. it takes HTML template files and turns them into tiny javascript functions). This can speed up template rendering on the client, and reduce bandwidth usage.
>
> [usage docs]

less

> Compiles LESS files into CSS. Only the `assets/styles/importer.less` is compiled. This allows you to control the ordering yourself, i.e. import your dependencies, mixins, variables, resets, etc. before other stylesheets.
>
> [usage docs]

sails-linker

> Automatically inject `<script>` tags for javascript files and `<link>` tags for css files. Also automatically links an output file containing precompiled templates using a `<script>` tag. A much more detailed description of this task can be found here, but the big takeaway is that script and stylesheet injection is *only* done in files containing `<!--SCRIPTS--><!--SCRIPTS END-->` and/or `<!--STYLES--><!--STYLES END-->` tags. These are included in the default **views/layout.ejs** file in a new Sails project. If you don't want to use the linker for your project, you can simply remove those tags.
>
> usage docs

sync

> A grunt task to keep directories in sync. It is very similar to grunt-contrib-copy but tries to copy only those files that have actually changed. It specifically synchronizes files from the `assets/` folder to `.tmp/public/`, overwriting anything that's already there.
>
> usage docs

uglify

> Minifies client-side javascript assets.
>
> usage docs

watch

> Runs predefined tasks whenever watched file patterns are added, changed or deleted. Watches for changes on files in the `assets/` folder, and re-runs the appropriate tasks (e.g. less and jst compilation). This allows you to see changes to your assets reflected in your app without having to restart the Sails server.
>
> usage docs

Disabling Grunt

To disable Grunt integration in Sails, simply delete your Gruntfile (and/or `tasks/` folder). You can also disable the Grunt hook. Just set the `grunt` property to `false` in `.sailsrc` hooks like this:

```
{
    "hooks": {
        "grunt": false
    }
}
```

Can I customize this for SASS, Angular, client-side Jade templates, etc?

Yep! Just replace the relevant grunt task in your `tasks/` directory, or add a new one. Like SASS for example.

If you still want to use Grunt for other purposes, but don't want any of the default web front-end stuff, just delete your project's assets folder and remove the front-end oriented tasks from the `grunt/register/` and `grunt/config/` folders. You can also run `sails new myCoolApi --no-frontend` to omit the assets folder and front-end-oriented Grunt tasks for future projects. You can also replace your `sails-generate-frontend` module with alternative community generators, or create your own. This allows `sails new` to create the boilerplate for native iOS apps, Android apps, Cordova apps, SteroidsJS apps, etc.

NOTE:

When removing the grunt hook above you must also specify the following in `.sailsrc` in order for your assets to be served, otherwise all assets will return a `404` .

```
{
    "paths": {
        "public": "assets"
    }
}
```

Task Automation

Overview

The `tasks/` directory contains a suite of Grunt tasks and their configurations.

Tasks are mainly useful for bundling front-end assets, (like stylesheets, scripts, & client-side markup templates) but they can also be used to automate all kinds of repetitive development chores, from browserify compilation to database migrations.

Sails bundles some default tasks for convenience, but with literally hundreds of plugins to choose from, you can use tasks to automate just about anything with minimal effort. If someone hasn't already built what you need, you can always author and publish your own Grunt plugin to npm!

> If you haven't used Grunt before, be sure to check out the Getting Started guide, as it explains how to create a Gruntfile as well as install and use Grunt plugins.

Asset pipeline

The asset pipeline is the place where you will organize the assets that will be injected into your views, and it can be found in the `tasks/pipeline.js` file. Configuring these assets is simple and uses grunt task file configuration and wildcard/glob/splat patterns. They are broken down into three sections.

CSS Files to Inject

This is an array of css files to be injected into your html as `<link>` tags. These tags will be injected between the `<!--STYLES--><!--STYLES END-->` comments in any view in which they appear.

Javascript Files to Inject

This is an array of Javascript files that gets injected into your html as `<script>` tags. These tags will be injected between the `<!--SCRIPTS--><!--SCRIPTS END-->` comments in any view in which they appear. The files get injected in the order they are in the array (i.e. you should place the path of dependencies before the file that depends on them.)

Template Files to Inject

This is an array of html files that will compiled to a jst function and placed in a jst.js file. This file then gets injected as a `<script>` tag in between the `<!--TEMPLATES--><!--TEMPLATES END-->` comments in your html.

> The same grunt wildcard/glob/splat patterns and task file configuration are used in some of the task configuration js files themselves if you would like to change those too.

Task configuration

Configured tasks are the set of rules your Gruntfile will follow when run. They are completely customizable and are located in the `tasks/config/` directory. You can modify, omit, or replace any of these Grunt tasks to fit your requirements. You can also add your own Grunt tasks- just add a `someTask.js` file in this directory to configure the new task, then register it with the appropriate parent task(s) (see files in `tasks/register/*.js`). Remember, Sails comes with a set of useful default tasks that are designed to get you up and running with no configuration required.

Configuring a custom task.

Configuring a custom task into your project is very simple and uses Grunt's config and task APIs to allow you to make your task modular. Let's go through a quick example of creating a new task that replaces an existing task. Let's say we want to use the Handlebars templating engine instead of the underscore templating engine that comes configured by default:

- The first step is to install the handlebars grunt plugin using the following command in your terminal:

```
npm install grunt-contrib-handlebars --save-dev
```

- Create a configuration file at `tasks/config/handlebars.js`. This is where we'll put our handlebars configuration.

```
// tasks/config/handlebars.js
// --------------------------------
// handlebar task configuration.

module.exports = function(grunt) {

  // We use the grunt.config api's set method to configure an
  // object to the defined string. In this case the task
  // 'handlebars' will be configured based on the object below.
  grunt.config.set('handlebars', {
    dev: {
      // We will define which template files to inject
      // in tasks/pipeline.js
      files: {
        '.tmp/public/templates.js': require('../pipeline').templateFilesToInject
      }
    }
  });

  // load npm module for handlebars.
  grunt.loadNpmTasks('grunt-contrib-handlebars');
};
```

- Replace the path to source files in asset pipeline. The only change here will be that handelbars looks for files with the extension .hbs while underscore templates can be in simple html files.

```
// tasks/pipeline.js
// --------------------------------
// asset pipeline

var cssFilesToInject = [
  'styles/**/*.css'
];

var jsFilesToInject = [
  'js/socket.io.js',
  'js/sails.io.js',
  'js/connection.example.js',
  'js/**/*.js'
];

// We change this glob pattern to include all files in
// the templates/ direcotry that end in the extension .hbs
var templateFilesToInject = [
  'templates/**/*.hbs'
];

module.exports = {
  cssFilesToInject: cssFilesToInject.map(function(path) {
    return '.tmp/public/' + path;
  }),
  jsFilesToInject: jsFilesToInject.map(function(path) {
    return '.tmp/public/' + path;
  }),
  templateFilesToInject: templateFilesToInject.map(function(path) {
    return 'assets/' + path;
  })
};
```

- Include the handlebars task into the compileAssets and syncAssets registered tasks. This is where the jst task was being used and we are going to replace it with the newly configured handlebars task.

```
// tasks/register/compileAssets.js
// --------------------------------
// compile assets registered grunt task

module.exports = function (grunt) {
  grunt.registerTask('compileAssets', [
    'clean:dev',
    'handlebars:dev',       // changed jst task to handlebars task
    'less:dev',
    'copy:dev',
    'coffee:dev'
  ]);
};

// tasks/register/syncAssets.js
// --------------------------------
// synce assets registered grunt task

module.exports = function (grunt) {
  grunt.registerTask('syncAssets', [
    'handlebars:dev',      // changed jst task to handlebars task
    'less:dev',
    'sync:dev',
    'coffee:dev'
  ]);
};
```

- Remove jst task config file. We are no longer using it so we can get rid of `tasks/config/jst.js`. Simply delete it from your project.

> Ideally you should delete it from your project and your project's node dependencies. This can be done by running this command in your terminal.
>
> ```
> npm uninstall grunt-contrib-jst --save-dev
> ```

Task triggers

In development mode, Sails runs the `default` task (`tasks/register/default.js`). This compiles LESS, CoffeeScript, and client-side JST templates, then links to them automatically from your app's dynamic views and static HTML pages.

In production, Sails runs the `prod` task (`tasks/register/prod.js`) which shares the same duties as `default`, but also minifies your app's scripts and stylesheets. This reduces your application's load time and bandwidth usage.

These task triggers are "basic" Grunt tasks located in the `tasks/register/` folder. Below, you'll find the complete reference of all task triggers in Sails, and the command which kicks them off:

`sails lift`

Runs the **default** task (`tasks/register/default.js`).

`sails lift --prod`

Runs the **prod** task (`tasks/register/prod.js`).

`sails www`

Runs the **build** task (`tasks/register/build.js`) that compiles all the assets to `www` subfolder instead of `.tmp/public` using relative paths in references. This allows serving static content with Apache or Nginx instead of relying on 'www middleware'.

`sails www --prod` (production)

Runs the **buildProd** task (`tasks/register/buildProd.js`) that does the same as **build** task but also optimizes assets.

You may run other tasks by specifying setting NODE_ENV and creating a task list in tasks/register/ with the same name. For example, if NODE_ENV is QA, sails will run tasks/register/QA.js if it exists.

Configuration

Overview

While Sails dutifully adheres to the philosophy of convention-over-configuration, it is important to understand how to customize those handy defaults from time to time. For almost every convention in Sails, there is an accompanying set of configuration options that allow you to adjust or override things to fit your needs. This section of the docs includes a complete reference of the configuration options available in Sails.

Sails apps can be configured programmatically, by specifying environment variables or command-line arguments, by changing the local or global `.sailsrc` files, or (most commonly) using the boilerplate configuration files conventionally located in the `config/` folder of new projects. The authoritative, merged-together configuration used in your app is available at runtime on the `sails` global as `sails.config`.

Standard configuration files (`config/*`)

A number of configuration files are included in new Sails apps by default. These boilerplate files include a number of inline comments, which are designed to provide a quick, on-the-fly reference without having to jump back and forth between the docs and your text editor.

In most cases, the top-level keys on the `sails.config` object (e.g. `sails.config.views`) correspond to a particular configuration file (e.g. `config/views.js`) in your app; however configuration settings may be arranged however you like across the files in your `config/` directory. The important part is the name (i.e. key) of the setting- not the file it came from.

For instance, let's say you add a new file, `config/foo.js`:

```
// config/foo.js
// The object below will be merged into `sails.config.blueprints`:
module.exports.blueprints = {
  shortcuts: false
};
```

For an exhaustive reference of individual configuration options, and the file they live in by default, check out the reference pages in this section, or take a look at "`config/`" in The Anatomy of a Sails App for a higher-level overview.

Environment-specific files (`config/env/*`)

Settings specified in the standard configuration files will generally be available in all environments (i.e. development, production, test, etc.). If you'd like to have some settings take effect only in certain environments, you can use the special environment-specific files and folders:

- Any files saved under the `/config/env/<environment-name>` folder will be loaded *only* when Sails is lifted in the `<environment-name>` environment. For example, files saved under `config/env/production` will only be loaded when Sails is lifted in production mode.
- Any files saved as `config/env/<environment-name>.js` will be loaded *only* when Sails is lifted in the `<environment-name>` environment, and will be merged on top of any settings loaded from the environment-specific subfolder. For example, settings in `config/env/production.js` will take precedence over those in the files in the `config/env/production` folder.

The `config/local.js` file

You may use the `config/local.js` file to configure a Sails app for your local environment (your laptop, for example). The settings in this file take precedence over all other config files except .sailsrc. Since they're intended only for local use, they should not be put under version control (and are included in the default `.gitignore` file for that reason). Use `local.js` to store local database settings, change the port used when lifting an app on your computer, etc.

See http://sailsjs.org/documentation/concepts/Configuration/localjsfile.html for more information.

Accessing `sails.config` in your app

The `config` object is available on the Sails app instance (`sails`). By default, this is exposed on the global scope during lift, and therefore available from anywhere in your app.

Example

```
// This example checks that, if we are in production mode, csrf is enabled.
// It throws an error and crashes the app otherwise.
if (sails.config.environment === 'production' && !sails.config.csrf) {
  throw new Error('STOP IMMEDIATELY ! CSRF should always be enabled in a production deplo
}
```

Custom Configuration

Sails recognizes many different settings, namespaced under different top level keys (e.g. `sails.config.sockets` and `sails.config.blueprints`). However you can also use `sails.config` for your own custom configuration (e.g. `sails.config.someProprietaryAPI.secret`).

Example

```
// config/linkedin.js
module.exports.linkedin = {
  apiKey: '...',
  apiSecret: '...'
};
```

```
// In your controller/service/model/hook/whatever:
// ...
var apiKey = sails.config.linkedin.apiKey;
var apiSecret = sails.config.linkedin.apiSecret;
// ...
```

Configuring the `sails` Command-Line Interface

When it comes to configuration, most of the time you'll be focused on managing the runtime settings for a particular app: the port, database connections, and so forth. However it can also be useful to customize the Sails CLI itself; to simplify your workflow, reduce repetitive tasks, perform custom build automation, etc. Thankfully, Sails v0.10 added a powerful new tool to do just that.

The `.sailsrc` file is unique from other configuration sources in Sails in that it may also be used to configure the Sails CLI-- either system-wide, for a group of directories, or only when you are `cd` 'ed into a particular folder. The main reason to do this is to customize the generators that are used when `sails generate` and `sails new` are run, but it can also be useful to install your own custom generators or apply hard-coded config overrides.

And since Sails will look for the "nearest" `.sailsrc` in the ancestor directories of the current working directory, you can safely use this file to configure sensitive settings you can't check in to your cloud-hosted code repository (*like your **database password**.*) Just include a `.sailsrc` file in your "$HOME" directory. See the docs on `.sailsrc` files for more information.

Notes

> The built-in meaning of the settings in `sails.config` are, in some cases, only interpreted by Sails during the "lift" process. In other words, changing some options at runtime will have no effect. To change the port your app is running on, for instance, you can't just change `sails.config.port` -- you'll need to change or override the setting in a configuration file or as a command-line argument, etc., then restart the server.

The `config/local.js` file

The `config/local.js` file is useful for configuring a Sails app for your local environment (your laptop, for example). The settings in this file take precedence over all other config files except .sailsrc. Since they're intended only for local use, they should not be put under version control (and are included in the default `.gitignore` file for that reason). Use `local.js` to store local database settings, change the port used when lifting an app on your computer, etc.

While you’re developing your app, this config file should include any settings specifically for your development computer or server (db passwords, etc.) If you're using git, note that `config/local.js` is included in the `.gitignore` in new Sails apps by default, and so it won't be checked into your repository when you commit.

When you’re ready to deploy your app in production, you can also use this file for configuration options on the server where it will be deployed. However, for server deployments, environment variables are usually preferable. You can also use command-line arguments and the `.sailsrc` file as alternatives to `config/local.js` for your local development configuration. See the overview for general information about Sails configuration.

> **Note:** This file is included in your .gitignore, so if you’re using git as a version control solution for your Sails app, keep in mind that this file won’t be committed to your repository! Good news is, that means you can specify configuration for your local machine in this file without inadvertently committing personal information (like database passwords) to the repo. Plus, this prevents other members of your team from commiting their local configuration changes on top of yours.

Using .sailsrc Files

In addition to the other methods of configuring your app, as of version 0.10, you can now specify configuration for one or more apps in `.sailsrc` file(s) (thanks to Dominic Tarr's excellent `rc` module). `rc` files are most useful for configuring the command-line and/or applying configuration settings to ALL of the Sails apps you run on your computer.

When the Sails CLI runs a command, it first looks for `.sailsrc` files (in either JSON or .ini format) in the current directory and in your home folder (i.e. `~/.sailsrc`) (every newly generated Sails app comes with a boilerplate `.sailsrc` file). Then it merges them in to its existing configuration.

> Actually, Sails looks for `.sailsrc` files in a few other places (following rc conventions). You can put a `.sailsrc` file at any of those paths. That said, stick to convention when you can- the best place to put a global `.sailsrc` file is in your home directory (i.e. `~/.sailsrc`).

Controllers

Overview

Controllers (the **C** in **MVC**) are the principal objects in your Sails application that are responsible for responding to *requests* from a web browser, mobile application or any other system capable of communicating with a server. They often act as a middleman between your models and views. For many applications, the controllers will contain the bulk of your project's business logic.

Actions

Controllers are comprised of a set of functions called *actions*. Action methods can be bound to routes in your application so that when a client requests the route, the bound method is executed to perform some business logic and (in most cases) generate a response. For example, the `GET /hello` route in your application could be bound to a method like:

```
function (req, res) {
  return res.send("Hi there!");
}
```

so that any time a web browser is pointed to the `/hello` URL on your app's server, the page displays the message "Hi there".

Where are Controllers defined?

Controllers are defined in the `api/controllers/` folder. You can put any files you like in that folder, but in order for them to be loaded by Sails as controllers, a file must *end* in `Controller.js`. By convention, Sails controllers are usually *Pascal-cased*, so that every word in the filename (including the first word) is capitalized: for example, `UserController.js`, `MyController.js` and `SomeGreatBigController.js` are all valid, Pascal-cased names.

You may organize your controllers into groups by saving them in subfolders of `api/controllers`, however note that the subfolder name *will become part of the Controller's identity* when used for routing (more on that in the "Routing" section below).

What does a Controller file look like?

A controller file defines a Javascript object whose keys are action names, and whose values are the corresponding action methods. Here’s a simple example of a full controller file:

```
module.exports = {
  hi: function (req, res) {
    return res.send("Hi there!");
  },
  bye: function (req, res) {
    return res.redirect("http://www.sayonara.com");
  }
};
```

This controller defines two actions: the “hi” responds to a request with a string message, while the “bye” action responds by redirecting to another web site. The `req` and `res` objects will be familiar to anyone who has used Express.js to write a web application. This is by design, as Sails uses Express under the hood to handle routing. Take special note, however, of the lack of a `next` argument for the actions. Unlike Express middleware methods, Sails controller actions should always be the last stop in the request chain--that is, they should always result in either a response or an error. While it is possible to use `next` in an action method, you are strongly encouraged to use policies instead wherever possible.

"Thin" Controllers

Most MVC frameworks recommend writing "thin" controllers, and while Sails is no exception (it is a good idea to keep your Sails controllers as simple as possible) it is also helpful to understand "why?"

Controller code is inherently dependent on some sort of trigger or event. In a backend framework like Sails, this event is almost always an incoming request. So if you write a bunch of code in one of your controller actions, it is not uncommon for that code's scope to be dependent on the "request context" (the `req` and `res` objects). Which is fine...until you want to use that code from a slightly different action, or from the command line.

So the goal of the "thin controller" philosophy is to encourage decoupling of reusable code from any related scope entanglements. In Sails, you can achieve this in a number of different ways, but the most common strategies for extrapolating code from controllers are:

- Write a custom model method to encapsulate some code that performs a particular task relating to a particular model
- Write a service as a function to encapsulate some code that performs a particular application-specific task
- If you find some code which is useful across multiple different applications (and you have time to do this), you should extract it into a node module. Then you can share it

across your organization, use it in future projects, or better yet, publish it on npm under a permissive open-source license for other developers to use and help maintain.

Generating controllers

You can use the Sails command line tool to quickly generate a controller, by typing:

```
$ sails generate controller <controller name> [action names separated by spaces...]
```

For example, if you run the following command:

```
$ sails generate controller comment create destroy tag like
info: Generated a new controller `comment` at api/controllers/CommentController.js!
```

Sails will generate `api/controllers/CommentController.js` :

```
/**
 * CommentController.js
 *
 * @description :: Server-side logic for managing comments.
 */

module.exports = {

  /**
   * CommentController.create()
   */
  create: function (req, res) {
    return res.json({
      todo: 'Not implemented yet!'
    });
  },

  /**
   * CommentController.destroy()
   */
  destroy: function (req, res) {
    return res.json({
      todo: 'Not implemented yet!'
    });
  },

  /**
   * CommentController.tag()
   */
  tag: function (req, res) {
    return res.json({
      todo: 'Not implemented yet!'
    });
  },

  /**
   * CommentController.like()
   */
  like: function (req, res) {
    return res.json({
      todo: 'Not implemented yet!'
    });
  }
};
```

Routing to Controllers

By default, Sails will create a blueprint action route for each action in a controller, so that a `GET` request to `/:controllerIdentity/:nameOfAction` will trigger the action. If the example controller in the previous section was saved as `api/controllers/SayController.js`, then the `/say/hi` and `/say/bye` routes would be made available by default whenever the app was lifted. If the controller was saved under the subfolder `/we`, then the routes would be `/we/say/hi` and `/we/say/bye`. See the blueprints documentation for more information about Sails' automatic route binding.

Besides the default routing, Sails allows you to manually bind routes to controller actions using the `config/routes.js` file. Some examples of when you might want to use explicit routes are:

- When you want to use separate actions to handle the same route path, based on the HTTP method (aka verb). The aforementioned **action blueprint** routes bind *all* request methods for a path to a given action, including `GET`, `POST`, `PUT`, `DELETE`, etc.
- When you want an action to be available at a custom URL (e.g. `PUT /login`, `POST /signup`, or a "vanity URL" like `GET /:username`)
- When you want to set up additional options for how the route should be handled (e.g. special CORS configuration)

To manually bind a route to a controller action in the `config/routes.js` file, you can use the HTTP verb and path (i.e. the **route address**) as the key, and the controller name + `.` + action name as the value (i.e. the **route target**).

For example, the following manual route will cause your app to trigger the `makeIt()` action in `api/controllers/SandwichController.js` whenever it receives a POST request to `/make/a/sandwich`:

```
'POST /make/a/sandwich': 'SandwichController.makeIt'
```

> **Note:**
>
> For controller files saved in subfolders, the subfolder is part of the controller identity:
>
> ```
> '/do/homework': 'stuff/things/HomeworkController.do'
> ```
>
> This will cause the `do()` action in `api/controllers/stuff/things/HomeworkController.js` to be triggered whenever `/do/homework` is requested.

A full discussion of manual routing is out of the scope of this doc--please see the routes documentation for a full overview of the available options.

Custom Responses

Overview

Sails v.10 allows for customizable server responses. Sails comes with a handful of the most common response types by default. They can be found in the `/api/responses` directory of your project. To customize these, simply edit the appropriate .js file.

As a quick example, consider the following controller action:

```
foo: function(req, res) {
   if (!req.param('id')) {
     res.status(400);
     res.view('400', {message: 'Sorry, you need to tell us the ID of the FOO you want!'})
   }
   ...
}
```

This code handles a bad request by sending a 400 error status and a short message describing the problem. However, this code has several drawbacks, primarily:

- It isn't *normalized*; the code is specific to this instance, and we'd have to work hard to keep the same format everywhere
- It isn't *abstracted*; if we wanted to use a similar approach elsewhere, we'd have to copy / paste the code
- The response isn't *content-negotiated*; if the client is expecting a JSON response, they're out of luck

Now, consider this replacement:

```
foo: function(req, res) {
   if (!req.param('id')) {
     res.badRequest('Sorry, you need to tell us the ID of the FOO you want!');
   }
   ...
}
```

This approach has many advantages:

- Error payloads are normalized
- Production vs. Development logging is taken into account
- Error codes are consistent

- Content negotiation (JSON vs HTML) is taken care of
- API tweaks can be done in one quick edit to the appropriate generic response file

Responses methods and files

Any `.js` script saved in the `/api/responses` folder will be executed by calling `res.[responseName]` in your controller. For example, `/api/responses/serverError.js` can be executed with a call to `res.serverError(errors)`. The request and response objects are available inside the response script as `this.req` and `this.res`; this allows the actual response function to take arbitrary parameters (like `serverError`'s `errors` parameter).

Adding a Custom Response

To add your own custom response method, simply add a file to `/api/responses` with the same name as the method you would like to create. The file should export a function, which can take any parameters you like.

```
/**
 * api/responses/myResponse.js
 *
 * This will be available in controllers as res.myResponse('foo');
 */

module.exports = function(message) {

  var req = this.req;
  var res = this.res;

  var viewFilePath = 'mySpecialView';
  var statusCode = 200;

  var result = {
    status: statusCode
  };

  // Optional message
  if (message) {
    result.message = message;
  }

  // If the user-agent wants a JSON response, send json
  if (req.wantsJSON) {
    return res.json(result, result.status);
  }

  // Set status code and view locals
  res.status(result.status);
  for (var key in result) {
    res.locals[key] = result[key];
  }
  // And render view
  res.render(viewFilePath, result, function(err) {
    // If the view doesn't exist, or an error occured, send json
    if (err) {
      return res.json(result, result.status);
    }

    // Otherwise, serve the `views/mySpecialView.*` page
    res.render(viewFilePath);
  });
```

Default responses

The following responses are bundled with all new Sails apps inside the `/api/responses` folder. Each one sends a normalized JSON object if the client is expecting JSON, containing a `status` key with the HTTP status code, and additional keys with relevant information about any errors.

res.serverError(errors)

This response normalizes the error/errors of `errors` into an array of proper, readable `Error` objects. `errors` can be one or more strings or `Error` objects. It then logs all Errors to the Sails logger (usually the console), and responds with the `views/500.*` view file if the client is expecting HTML, or a JSON object if the client is expecting JSON. In development mode, the list of errors is included in the response. In production mode, the actual errors are suppressed.

res.badRequest(validationErrors, redirectTo)

For requesters expecting JSON, this response includes the 400 status code and any relevant data sent as `validationErrors`.

For traditional (not-AJAX) web forms, this middleware follows best-practices for when a user submits invalid form data:

- First, a one-time-use flash variable is populated, probably a string message or an array of semantic validation error objects.
- Then the user is redirected back to `redirectTo`, i.e. the URL where the bad request originated.
- There, the controller and/or view might use the flash `errors` to either display a message or highlight the invalid HTML form fields.

res.notFound()

If the requester is expecting JSON, this response simply sends a 404 status code and a `{status: 404}` object.

Otherwise the view located in `myApp/views/404.*` will be served. If that view can't be found, then the client is just sent the JSON response.

res.forbidden(message)

If the requester is expecting JSON, this response sends the 403 status code along with the contents of `message`.

Otherwise the view located in `myApp/views/403.*` will be served. If that view can't be found, then the client is just sent the JSON response.

Deployment

Overview

Before You Deploy

Before you launch any web application, you should ask yourself a few questions:

- What is your expected traffic?
- Are you contractually required to meet any uptime guarantees, e.g. a Service Level Agreement (SLA)?
- What sorts of front-end apps will be "hitting" your infrastructure?
    - Android apps
    - iOS apps
    - desktop web browsers
    - mobile web browsers (tablets, phones, iPad minis?)
    - tvs, watches, toasters..?
- And what kinds of things will they be requesting?
    - JSON?
    - HTML?
    - XML?
- Will you be taking advantage of realtime pubsub features with Socket.io?
    - e.g. chat, realtime analytics, in-app notifications/messages
- How are you tracking crashes and errors?
    - Take a look at Sails' log config

Deploying On a Single Server

Node.js is pretty darn fast. For many apps, one server is enough to handle the expected traffic-- at least at first.

Configure

- All your production environment settings are stored in `config/env/production.js`
- Configure your app to run on port 80 (if not behind a proxy like nginx). If you're using nginx, be sure to configure it to relay websockets to your app. You can find guidance here in nginx docs WebSocket proxying.
- Configure the 'production' environment so that all of your css/js gets bundled up, and the internal servers are switched into the appropriate environment (requires linker)

- Make sure your database is set-up on the production server. This is especially important if you are using a relational database such as MySQL, because sails sets all your models to `migrate:safe` when run in production, which means no auto-migrations are run on starting up the app. You can set your database up the following way:
  - Create the database on the server and then run your sails app with `migrate:alter` locally, but configured to use the production server as your db. This will automatically set things up.
  - In case you can't connect to the server remotely, you'll simply dump your local schema and import it into the database server.
- Enable CSRF protection for your POST, PUT, and DELETE requests
- Enable SSL
- IF YOU'RE USING SOCKETS:
  - Configure `config/sockets.js` to use socket.io's recommended production settings here
    - e.g. enable the `flashsocket` transport

Deploy

In production, instead of `sails lift`, you'll want to use forever or PM2 to make sure your app will keep running, even if it crashes.

- Install forever: `sudo npm install -g forever`
  - More about forever: https://github.com/nodejitsu/forever
- Or install PM2: `sudo npm install pm2 -g --unsafe-perm`
  - More information about that: https://github.com/Unitech/pm2
- From your app directory, start the server either with `forever start app.js --prod` or `pm2 start app.js -x -- --prod`
  - This is the same thing as using `sails lift --prod`, but if the server crashes, it will be automatically restarted.

FAQ

Can I use environment variables?

Yes! You can also configure the `port` and `environment` settings in Sails using environment variables. `NODE_ENV=production sails lift` `PORT=443 sails lift`

Where do I put my production database credentials? Other settings?

For your other deployment/machine-specific settings, namely any kind of credentials, you should use `config/local.js`.
It's included in your `.gitignore` file by default so you don't inadvertently commit your credentials to your code repository.

config/local.js

```
// Local configuration
//
// Included in the .gitignore by default,
// this is where you include configuration overrides for your local system
// or for a production deployment.
//
// For example, to use port 80 on the local machine, override the `port` config
module.exports = {
    port: 80,
    environment: 'production',
    adapters: {
        mysql: {
            user: 'root',
            password: '12345'
        }
    }
}
```

How do I get my Sails app on the server?

Is your Node.js instance already spun up? When you have the ip address, you can go ahead and ssh onto it, then `sudo npm install -g forever` to install Sails and forever for the first time.

Then `git clone` your project (or `scp` it onto the server if it's not in a git repo) into a new folder on the server and cd into it, and `forever start app.js`

Performance Benchmarks

Performance in Sails is comparable to what you'd expect from a standard Node.js/Express application. In other words, fast! We've done some optimizations ourselves in Sails and Waterline, but primarily, our focus has been on not messing up what was already really fast. Above all, we have @ry, @visionmedia, @isaacs, #v8, @joyent and the rest of the Node.js core team to thank.

- http://serdardogruyol.com/?p=111

Issues #3099 and #2779 are about a memory leak. It resides in the `express-session` module used by default, which stores sessions in-memory. To disable it, make sure that you disable sessions in the `.sailsrc` :

```
"hooks": {
  "session": false
}
```

You may also use an alternative (redis/mongo/cookies) to store sessions.

Hosting

Here is a non-comprehensive list of Sails.js hosting providers.

Deploying to Modulus?

- http://blog.modulus.io/sails-js

Deploying to OpenShift?

To deploy to OpenShift, you'll need to make some minor modifications to your configuration: Open up `config/local.js` in your app folder. In here, you'll need to add the following lines.

```
    port: process.env.OPENSHIFT_NODEJS_PORT,
    host: process.env.OPENSHIFT_NODEJS_IP,
```

You will also need to install `grunt-cli` with `npm i --save grunt-cli`.

After doing that, create the file `.openshift/action_hooks/pre_start_nodejs` with the following contents. (source)

```
#!/bin/bash
export NODE_ENV=production

if [ -f "${OPENSHIFT_REPO_DIR}"/Gruntfile.js ]; then
    (cd "${OPENSHIFT_REPO_DIR}"; node_modules/grunt-cli/bin/grunt prod)
fi
```

Then create the file `/supervisor_opts` with the following contents. This tells OpenShift's supervisor to ignore Sails' `.tmp` directory for the hot reload functionality. (source)

```
-i .tmp
```

You can now `git add . && git commit -a -m "your message" && git push` to deploy to OpenShift.

Using DigitalOcean?

- https://www.digitalocean.com/community/articles/how-to-create-an-node-js-app-using-sails-js-on-an-ubuntu-vps
- https://www.digitalocean.com/community/articles/how-to-use-pm2-to-setup-a-node-js-production-environment-on-an-ubuntu-vps

- https://www.digitalocean.com/community/articles/how-to-host-multiple-node-js-applications-on-a-single-vps-with-nginx-forever-and-crontab

Deploying to Heroku?

- Sails.js and Heroku
- SailsCasts: Deploying a Sails App to Heroku
- Sails.js on Heroku
- https://groups.google.com/forum/#!topic/sailsjs/vgqJFr7maSY
- https://github.com/chadn/heroku-sails
- http://dennisrongo.com/deploying-sails-js-to-heroku
- http://stackoverflow.com/a/20184907/486547

Deploying to AWS?

- http://blog.grio.com/2014/01/your-own-mini-heroku-on-aws.html
- http://serverfault.com/questions/531560/creating-an-sails-js-application-on-aws-ami-instance
- http://bussing-dharaharsh.blogspot.com/2013/08/creating-sailsjs-application-on-aws-ami.html
- http://cloud.dzone.com/articles/how-deploy-nodejs-apps-aws-mac

Using PM2?

- http://devo.ps/blog/goodbye-node-forever-hello-pm2/

Deploying to CloudControl?

- https://www.cloudcontrol.com/dev-center/Guides/NodeJS/Sailsjs

Getting professional help

These days, it's getting easier and easier to deploy powerful applications at scale. That said, there isn't always time to do these things yourself. Sails.js is maintained by my company, Balderdash, a Node.js consultancy in Austin, TX. If your company needs professional support, reach out and we're happy to help. The deployment part really isn't that hard, and in most cases, it shouldn't take more than a couple of hours tops.

Scaling

If you have the immediate expectation of lots of traffic to your application (or better yet, you already have it!), you'll want to set up a scalable architecture that your app can scale as more and more people use it.

Benchmarks

For the most part, Sails benchmarks exactly like any Connect, Express or Socket.io app. This has been validated on a few different occasions, most recently here. If you have your own benchmark you'd like to share, please send a pull request to this page on Github.

Example architecture

```
                         Sails.js server
                               ....
                      /  Sails.js server  \      /  Database (e.g. Mongo, Postgres, etc)
Load Balancer  <-->      Sails.js server    <-->    Socket store (Redis)
                      \  Sails.js server  /      \  Session store (Redis)
                               ....
                         Sails.js server
```

Configuring your app for a clustered deployment

- Make sure the database(s) for your models (e.g. MySQL, Postgres, Mongo) is scalable (e.g. sharding/cluster)
- Configure your app to use a shared session store
  - Support for redis is built in (see the `adapter` options in `config/session.js`)
- IF YOU'RE USING SOCKETS:
  - Configure your app to use a shared socket store
    - Support for redis is built in (see the `adapter` options in `config/sockets.js`)
    - The default Socket.io configuration initially attempts to connect to the server using long-polling. In order for this to work, your server environment must support sticky load-balancing (aka sticky sessions), otherwise the handshake will fail until the connection is upgraded to use Websockets (and only if Websockets are available). On **Heroku**, you must have the sticky load-balancing beta feature explicitly enabled. In an environment without stickky load balancing, you will need to set the `transports` setting in config/sockets.js to `['websocket']` , forcing it to use websockets only and avoid long-polling.

You'll also need to set the transports in your socket client--if you're using `sails.io.js`, this is as easy as adding a `<script>io.sails.transports=['websocket']</script>` immediately after the `sails.io.js` script include. For a rather dramatic read on the issue, see this thread.

- Ensure none of the other dependencies you might be using in your app rely on shared memory.

Deploying a Sails cluster on multiple servers

- Deploy multiple instances (aka servers running a copy of your app) behind a load balancer
    - Start up Sails on each instance using `forever`
    - More on load balancers: https://en.wikipedia.org/wiki/Load_balancing_(computing)
- Configure your load balancer to terminate SSL requests
    - Because of this, you won't need to use the SSL configuration in Sails-- the traffic will already be decrypted

File Uploads

Uploading files in Sails is similar to how you would upload files for a vanilla Node.js or Express application. It is, however, probably different than what you're used to if you're coming from a different server-side platform like PHP, .NET, Python, Ruby, or Java. But fear not: the core team has gone to great lengths to make file uploads easier to accomplish, while still keeping them scalable and secure.

Sails comes with a powerful "body parser" called Skipper which makes it easy to implement streaming file uploads-- not only to the server's filesystem (i.e. hard disk), but also to Amazon S3, MongoDB's gridfs, or any of its other supported file adapters.

Uploading a file

Files are uploaded to HTTP web servers as *file parameters*. In the same way you might send a form POST to a URL with text parameters like "name", "email", and "password", you send files as file parameters, like "avatar" or "newSong".

Take this simple example:

```
req.file('avatar').upload(function (err, uploadedFiles) {
  // ...
});
```

Files should be uploaded inside of an `action` in one of your controllers. Here's a more in-depth example that demonstrates how you could allow users to upload an avatar image and associate it with their accounts. It assumes you've already taken care of access control in a policy, and that you're storing the id of the logged-in user in `req.session.me` .

```
// api/controllers/UserController.js
//
// ...


/**
 * Upload avatar for currently logged-in user
 *
 * (POST /user/avatar)
 */
uploadAvatar: function (req, res) {

  req.file('avatar').upload({
    // don't allow the total upload size to exceed ~10MB
```

```
    maxBytes: 10000000
  },function whenDone(err, uploadedFiles) {
    if (err) {
      return res.negotiate(err);
    }

    // If no files were uploaded, respond with an error.
    if (uploadedFiles.length === 0){
      return res.badRequest('No file was uploaded');
    }


    // Save the "fd" and the url where the avatar for a user can be accessed
    User.update(req.session.me, {

      // Generate a unique URL where the avatar can be downloaded.
      avatarUrl: require('util').format('%s/user/avatar/%s', sails.getBaseUrl(), req.sess

      // Grab the first file and use it's `fd` (file descriptor)
      avatarFd: uploadedFiles[0].fd
    })
    .exec(function (err){
      if (err) return res.negotiate(err);
      return res.ok();
    });
  });
},


/**
 * Download avatar of the user with the specified id
 *
 * (GET /user/avatar/:id)
 */
avatar: function (req, res){

  req.validate({
    id: 'string'
  });

  User.findOne(req.param('id')).exec(function (err, user){
    if (err) return res.negotiate(err);
    if (!user) return res.notFound();

    // User has no avatar image uploaded.
    // (should have never have hit this endpoint and used the default image)
    if (!user.avatarFd) {
      return res.notFound();
    }

    var SkipperDisk = require('skipper-disk');
    var fileAdapter = SkipperDisk(/* optional opts */);
```

```
    // Stream the file down
    fileAdapter.read(user.avatarFd)
    .on('error', function (err){
      return res.serverError(err);
    })
    .pipe(res);
  });
}

//
// ...
```

Where do the files go?

When using the default `receiver`, file uploads go to the `myApp/.tmp/uploads/` directory. You can override this using the `dirname` option. Note that you'll need to provide this option both when you call the `.upload()` function AND when you invoke the skipper-disk adapter (so that you are uploading to and downloading from the same place.)

Uploading to a custom folder

In the above example we upload the file to .tmp/uploads. So how do we configure it with a custom folder, say ‘assets/images’. We can achieve this by adding options to upload function as shown below.

```
req.file('avatar').upload({
  dirname: require('path').resolve(sails.config.appPath, '/assets/images')
},function (err, uploadedFiles) {
  if (err) return res.negotiate(err);

  return res.json({
    message: uploadedFiles.length + ' file(s) uploaded successfully!'
  });
});
```

Example

Generate an `api`

First we need to generate a new `api` for serving/storing files. Do this using the sails command line tool.

```
$ sails generate api file

debug: Generated a new controller `file` at api/controllers/FileController.js!
debug: Generated a new model `File` at api/models/File.js!

info: REST API generated @ http://localhost:1337/file
info: and will be available the next time you run `sails lift`.
```

Write Controller Actions

Lets make an `index` action to initiate the file upload and an `upload` action to receive the file.

```
// myApp/api/controllers/FileController.js

module.exports = {

  index: function (req,res){

    res.writeHead(200, {'content-type': 'text/html'});
    res.end(
    '<form action="http://localhost:1337/file/upload" enctype="multipart/form-data" metho
    '<input type="text" name="title"><br>'+
    '<input type="file" name="avatar" multiple="multiple"><br>'+
    '<input type="submit" value="Upload">'+
    '</form>'
    )
  },
  upload: function  (req, res) {
    req.file('avatar').upload(function (err, files) {
      if (err)
        return res.serverError(err);

      return res.json({
        message: files.length + ' file(s) uploaded successfully!',
        files: files
      });
    });
  }

};
```

Where do they go?

When using the default `receiver`, file uploads go to the `myApp/.tmp/uploads/` directory. You can do whatever you want with it in the `upload` action.

Uploading to a custom folder

In the above example we could upload the file to .tmp/uploads . So how do we configure it with a custom folder , say ‘assets/images’. We can achieve this by adding options to upload function as shown below.

```
req.file('avatar').upload({
  dirname: './assets/images'
},function (err, uploadedFiles) {
  if (err) return res.negotiate(err);

  return res.json({
    message: uploadedFiles.length + ' file(s) uploaded successfully!'
  });
});
```

Read more

- Skipper docs
- Uploading to Amazon S3
- Uploading to Mongo GridFS

Uploading to Amazon S3

> Please note that your Amazon S3 bucket must be created in the 'US Standard' region. If you fail to do so, you will get a 'TypeError('Uncaught, unspecified "error" event.').

With Sails, you can stream file uploads to Amazon S3 with very little additional configuration.

First install the S3 Skipper adapter:

```
$ npm install skipper-s3 --save
```

Then use it in one of your controllers:

```
  uploadFile: function (req, res) {
    req.file('avatar').upload({
      adapter: require('skipper-s3'),
      key: 'S3 Key'
      secret: 'S3 Secret'
      bucket: 'Bucket Name'
    }, function (err, filesUploaded) {
      if (err) return res.negotiate(err);
      return res.ok({
        files: filesUploaded,
        textParams: req.params.all()
      });
    });
  }
```

Uploading to Mongo GridFS

Uploading files to MongoDB is possible thanks to Mongo's GridFS filesystem. With Sails, you can accomplish this with very little additional configuration using the Skipper adapter for MongoDB's GridFS.

Install it with:

```
$ npm install skipper-gridfs --save
```

Then use it in one of your controllers:

```
  uploadFile: function (req, res) {
    req.file('avatar').upload({
      adapter: require('skipper-gridfs'),
      uri: 'mongodb://[username:password@]host1[:port1][/[database[.bucket]]'
    }, function (err, filesUploaded) {
      if (err) return res.negotiate(err);
      return res.ok();
    });
  }
```

Globals

Overview

For convenience, Sails exposes a handful of global variables. By default, your app's models, services, and the global `sails` object are all available on the global scope; meaning you can refer to them by name anywhere in your backend code (as long as Sails has been loaded).

Nothing in Sails core relies on these global variables - each and every global exposed in Sails may be disabled in `sails.config.globals` (conventionally configured in `config/globals.js`.)

The App Object (`sails`)

In most cases, you will want to keep the `sails` object globally accessible- it makes your app code much cleaner. However, if you *do* need to disable *all* globals, including `sails`, you can get access to `sails` on the request object (`req`).

Models and Services

Your app's models and services are exposed as global variables using their `globalId`. For instance, the model defined in the file `api/models/Foo.js` will be globally accessible as `Foo`, and the service defined in `api/services/Baz.js` will be available as `Baz`.

Async (`async`) and Lodash (`_`)

Sails also exposes an instance of lodash as `_`, and an instance of async as `async`. These commonly-used utilities are provided by default so that you don't have to `npm install` them in every new project. Like any of the other globals in sails, they can be disabled.

Disabling Globals

Sails determines which globals to expose by looking at `sails.config.globals`, which is conventionallly configured in `config/globals.js`.

To disable all global variables, just set the setting to `false`:

```
// config/globals.js
module.exports.globals = false;
```

To disable *some* global variables, specify an object instead, e.g.:

```
// config/globals.js
module.exports.globals = {
  _: false,
  async: false,
  models: false,
  services: false
};
```

Notes

> - Bear in mind that none of the globals, including `sails`, are accessible until *after* sails has loaded. In other words, you won't be able to use `sails.models.user` or `User` outside of a function (since `sails` will not have finished loading yet.)

Internationalization

Overview

If your app will touch people or systems from all over the world, internationalization and localization (i18n) may be an important part of your international strategy. Sails provides built-in support for detecting user language preferences and translating static words/sentences thanks to i18n-node. (npm).

Usage

In a view:

```
<%= __('Hello')="" %="">

<%= __('Hello="" %s,="" how="" are="" y

<%= i18n('That\'s="" right--="" you="" can="" use="" either="" i18n()="" or="" __()')=""
```

In a controller or policy:

```
req.__('Hello'); // => Hola
req.__('Hello %s', 'Marcus'); // => Hola Marcus
req.__('Hello {{name}}', { name: 'Marcus' }); // => Hola Marcus
```

Or if you already know the locale id, you can translate from anywhere in your application using `sails.__` :

```
sails.__({
  phrase: 'Hello',
  locale: 'es'
});
// => 'Hola!'
```

Additional Options

Settings for localization/internationalization may be configured in `sails.config.i18n`. The most common reason you'll need to modify these settings is to edit the list of your app's supported locales and/or the location of your translation stringfiles:

```
// Which locales are supported?
locales: ['en', 'es'],

// Where are your locale translations located?
localesDirectory: '/config/locales'
```

Disabling or customizing Sails' default internationalization support

Of course you can always `require()` any Node modules you like, anywhere in your project, and use any internationalization strategy you want.

But worth noting is that since Sails implements node-i18n integration in the i18n hook, you can completely disable or override it using the `loadHooks` and/or `hooks` configuration options.

What About i18n on the client?

The above technique works great out of the box for server-side views. But what about rich client apps that serve static HTML templates from a CDN or static host? (e.g. performance-obsessed SPAs or PhoneGap apps/Chrome extensions)

You can actually reuse Sails' i18n support to help you get your translated templates to the browser. If you want to use Sails to internationalize your *client-side templates*, put your front-end templates in a subdirectory of your app's `/views` folder.

- In development mode, you should retranslate and precompile your templates each time the relevant stringfile or template changes using grunt-contrib-watch, which is already installed by default in new Sails projects.
- In production mode, you'll want to translate and precompile all templates on lift(). In loadtime-critical scenarios (e.g. mobile web apps) you can even upload your translated,

precompiled, minified templates to a CDN like Cloudfront for further performance gains.

Locales

Overview

The i18n hook reads JSON-formatted translation files from your project's "locales" directory (`config/locales` by default). Each file corresponds with a locale (usually a language) that your Sails backend will support.

These files contain locale-specific strings (as JSON key-value pairs) that you can use in your views, controllers, etc.

Here is an example locale file (`config/locales/es.json`):

```
{
    "Hello!": "Hola!",
    "Hello %s, how are you today?": "¿Hola %s, como estas?"
}
```

Note that the keys in your stringfiles (e.g. "Hello %s, how are you today?") are **case sensitive** and require exact matches. There are a few different schools of thought on the best approach here, and it really depends on who/how often you'll be editing the stringfiles vs. HTML in the future. Especially if you'll be editing the translations by hand, simpler, all-lowercase key names may be preferable for maintainability.

For example, here's another pass at `config/locales/es.json` :

```
{
    "hello": "Hola!",
    "hello-how-are-you-today": "Hola %s, ¿cómo estás?"
}
```

And here's `config/locales/en.json` :

```
{
    "hello": "Hello!",
    "hello-how-are-you-today": "Hello, how are you today?"
}
```

You can also nest locale strings. But a better approach would be to use `.` to represent nested strings. For example, here's the list of labels for the index page of a user controller:

```
{
    "user.index.label.id": "User ID",
    "user.index.label.name": "User Name"
}
```

Detecting and/or overriding the desired locale for a request

To determine the current locale used by the request, use `req.getLocale()` .

To override the auto-detected language/localization preference for a request, use `req.setLocale()` , calling it with the unique code for the new locale, e.g.:

```
// Force the language to German for the remainder of the request:
req.setLocale('de');
// (this will use the strings located in `config/locales/de.json` for translation)
```

By default, node-i18n will detect the desired language of a request by examining its language headers. Language headers are set in your users' browser settings, and while they're correct most of the time, you may need the flexibility to override this detected locale and provide your own.

For instance, if your app allows users to pick their preferred language, you might create a policy which checks for a custom language in the user's session, and if one exists, sets the appropriate locale for use in subsequent policies, controller actions, and views:

```
// api/policies/localize.js
module.exports = function(req, res, next) {
  req.setLocale(req.session.languagePreference);
  next();
};
```

Translating Dynamic Content

If your backend is storing interlingual data (e.g. product data is entered in multiple languages via a CMS), you shouldn't rely on simple JSON locale files unless you're somehow planning on editing your locale translations dynamically. One option is to edit the locale translations programatically, either with a custom implementation or through a translation service. Sails/node-i18n JSON stringfiles are compatible with the format used by webtranslateit.com.

On the other hand you might opt to store these types of dynamic translated strings in a database. If so, just make sure and build your data model accordingly so you can store and retrieve the relevant dynamic data by locale id (e.g. "en", "es", "de", etc) That way, you can leverage the `req.getLocale()` method to help you figure out which translated content to use in any given response, and keep consistent with the conventions used elsewhere in your app.

Logging

Overview

Sails comes with a simple, built-in logger called `captains-log`. Its usage is purposely very similar to Node's `console.log`, but with a handful of extra features; namely support for multiple log levels with colorized, prefixed console output.

Configuration

The Sails logger's configuration is located in `sails.config.log`, for which a conventional configuration file (`config/log.js`) is bundled in new Sails projects out of the box.

When configured at a given log level, Sails will output log messages that are output at a level at or above the currently configured level. This log level is normalized and also applied to the generated output from socket.io, Waterline, and other dependencies. The hierarchy of log levels and their relative priorities is summarized by the chart below:

Priority	level	Log fns visible
0	silent	N/A
1	error	`.error()`
2	warn	`.warn()` , `.error()`
3	debug	`.debug()` , `.warn()` , `.error()`
4	info	`.info()` , `.debug()` , `.warn()` , `.error()`
5	verbose	`.verbose()` , `.info()` , `.debug()` , `.warn()` , `.error()`
6	silly	`.silly()` , `.verbose()` , `.info()` , `.debug()` , `.warn()` , `.error()`

Notes

- The default log level is "info". When your app's log level is set to "info", Sails logs limited information about the server/app's status.
- When the log level is set to "silly", Sails outputs internal information on which routes are being bound and other detailed framework lifecycle information, diagnostics, and implementation details.
- When the log level is set to "verbose", Sails logs Grunt output, as well as much more detailed information on the routes, models, hooks, etc. that were loaded.

sails.log()

Overview

Each of the methods below accepts an infinite number of arguments of any data type, seperated by commas. Like `console.log`, data passed as arguments to the Sails logger is automatically prettified for readability using Node's `util.inspect()`. Consequently, standard Node.js conventions apply- i.e. if you log an object with an `inspect()` method, it will be run automatically, and the string that it returns will be written to the console. Similarly, objects, dates, arrays, and most other data types are pretty-printed using the built-in logic in `util.inspect()` (e.g. you see `{ pet: { name: 'Hamlet' } }` instead of `[object Object]`.)

`sails.log()`

The default log function, which writes console output to `stderr` at the "debug" log level.

```
sails.log('hello');
// -> debug: hello.
```

`sails.log.error()`

Writes log output to `stderr` at the "error" log level.

```
sails.log.error('Unexpected error occurred.');
// -> error: Unexpected error occurred.
```

`sails.log.warn()`

Writes log output to `stderr` at the "warn" log level.

```
sails.log.warn('File upload quota exceeded for user','request aborted.');
// -> warn: File upload quota exceeded for user- request aborted.
```

`sails.log.debug()`

Alias for `sails.log()`

`sails.log.info()`

Writes log output to `stdout` at the "info" log level.

```
sails.log.info('A new user (', 'mike@foobar.com', ') just signed up!');
// -> info: A new user ( mike@foobar.com ) just signed up!
```

`sails.log.verbose()`

Writes log output to `stdout` at the "verbose" log level. Useful for capturing detailed information about your app that you might only want to enable on rare occasions.

```
sails.log.verbose('A user initiated an account transfer...')
// -> verbose: A user initiated an account transfer...
```

`sails.log.silly()`

Writes log output to `stdout` at the "silly" log level. Useful for capturing utterly ridiculous information about your app you only need on rare occasions.

```
sails.log.silly('A user probably clicked on something..?');
// -> silly: A user probably clicked on something..?
```

Middleware

Sails is fully compatible with Express / Connect middleware - in fact, it's all over the place! Much of the code you'll write in Sails is effectively middleware; most notably controller actions and policies.

HTTP Middleware

Sails also utilizes an additional configurable middleware stack just for handling HTTP requests. Each time your app receives an HTTP request, the configured HTTP middleware stack runs in order.

> Note that this HTTP middleware stack is only used for "true" HTTP requests-- it is ignored for **virtual requests** (e.g. requests from a live Socket.io connection.)

Legend:

- `*` - The middleware with an asterisk (*) above should *almost never* need to be modified or removed. Please only do so if you really understand what you're doing.

Adding or Overriding HTTP Middleware

To configure a custom HTTP middleware function, define a new HTTP key `sails.config.http.middleware.foobar` and set it to the configured middleware function, then add the string name ("foobar") to your `sails.config.http.middleware.order` array wherever you'd like it to run in the middleware chain (a good place to put it might be right before "cookieParser"):

E.g. in `config/http.js` :

```
// ...
middleware: {

  // Define a custom HTTP middleware fn with the key `foobar`:
  foobar: function (req,res,next) { /*...*/ next(); },

  // Define another couple of custom HTTP middleware fns with keys `passportInit` and `
  // (notice that this time we're using an existing middleware library from npm)
  passportInit    : require('passport').initialize(),
  passportSession : require('passport').session(),

  // Override the conventional cookie parser:
  cookieParser: function (req, res, next) { /*...*/ next(); },


  // Now configure the order/arrangement of our HTTP middleware
  order: [
    'startRequestTimer',
    'cookieParser',
    'session',
    'passportInit',            // <==== passport HTTP middleware should run after "sess
    'passportSession',         // <==== (see https://github.com/jaredhanson/passport#mi
    'bodyParser',
    'compress',
    'foobar',                  // <==== we can put this stuff wherever we want
    'methodOverride',
    'poweredBy',
    '$custom',
    'router',
    'www',
    'favicon',
    '404',
    '500'
  ]
},

customMiddleware: function(app){
   //Intended for other middleware that doesn't follow 'app.use(middleware)' convention
   require('other-middleware').initialize(app);
}
// ...
```

Express Middleware In Sails

One of the really nice things about Sails apps is that they can take advantage of the wealth of already-existing Express/Connect middleware out there. But a common question that arises when people *actually* try to do this is:

> *"Where do I `app.use()` this thing?".*

In most cases, the answer is to install the Express middleware as a custom HTTP middleware in `sails.config.http.middleware`. This will trigger it for ALL HTTP requests to your Sails app, and allow you to configure the order in which it runs in relation to other HTTP middleware.

Express Routing Middleware In Sails

You can also include Express middleware as a policy- just configure it in `config/policies.js`. You can either require and setup the middleware in an actual wrapper policy (usually a good idea) or just require it directly in your policies.js file. The following example uses the latter strategy for brevity:

```
{
  '*': true,

  ProductController: {

    // Prevent end users from doing CRUD operations on products reserved for admins
    // (uses HTTP basic auth)
    '*': require('http-auth')({
      realm: 'admin area'
    }, function customAuthMethod (username, password, onwards) {
      return onwards(username === "Tina" && password === "Bullock");
    }),

    // Everyone can view product pages
    show: true
  }
}
```

Conventional Defaults

Sails comes bundled with a suite of conventional HTTP middleware, ready to use. You can, of course, disable, override, rearrange, or append to it, but the pre-installed stack is perfectly acceptable for most apps in development or production. Below is a list of the standard HTTP middleware functions that comes bundled in Sails in the order they execute every time the server receives an incoming HTTP request:

HTTP Middleware Key	Purpose
startRequestTimer	Allocates a variable in memory to hold the timestamp when the request began. This can be accessed and used by your app to provide diagnostic information about slow requests.
*cookieParser **	Parses the cookie header into a clean object for use in subsequent middleware and your application code.
*session **	Sets up a unique session object using your session configuration.
bodyParser	Parses parameters and binary upstreams (for streaming file uploads) from the HTTP request body using Skipper.
compress	Compresses response data using gzip/deflate.
methodOverride	Provides faux HTTP method support, letting you use HTTP verbs such as PUT or DELETE in places where the client doesn't support it (e.g. legacy versions of Internet Explorer.) If a request has a `_method` parameter set to `"PUT"` , the request will be routed as if it was a proper PUT request. See Connect's methodOverride docs for more information if you need it.
poweredBy	Attaches an `X-Powered-By` header to outgoing responses.
$custom	Provides backwards compatibility for a configuration option from Sails v0.9.x. Since Sails v0.10 offers much more configuration flexibility for HTTP middleware, as long as you are not using `sails.config.express.customMiddleware` , you can confidently remove this item from the list.
*router **	This is where the bulk of your app logic gets applied to any given request. In addition to running `"before"` handlers in hooks (e.g. csrf token enforcement) and some internal Sails logic, this routes requests using your app's explicit routes (in `sails.config.routes`) and/or route blueprints.
*www **	Serves static files- usually images, stylesheets, scripts- in your app's "public" folder (configured in `sails.config.paths` , conventionally `.tmp/public/`) using Connect's static middleware.
favicon	Serves the browser favicon for your app if one is provided as `/assets/favicon.ico` .
*404 **	Handles requests which do not match any routes - triggers `res.notFound()`
*500 **	Handles requests which trigger an internal error (i.e. call Express's `next(err)`) - triggers `res.serverError()`

Waterline: SQL/noSQL Data Mapper (ORM/ODM)

Sails comes installed with a powerful ORM/ODM called Waterline, a datastore-agnostic tool that dramatically simplifies interaction with one or more databases. It provides an abstraction layer on top of the underlying database, allowing you to easily query and manipulate your data *without* writing vendor-specific integration code.

Database Agnosticism

In schemaful databases like Postgres, Oracle, and MySQL, models are represented by tables. In MongoDB, they're represented by Mongo "collections". In Redis, they're represented using key/value pairs. Each database has its own distinct query dialect, and in some cases even requires installing and compiling a specific native module to connect to the server. This involves a fair amount of overhead, and garners an unsettling level of vendor lock-in to a specific database; e.g. if your app uses a bunch of SQL queries, it will be very hard to switch to Mongo later, or Redis, and vice versa.

Waterline query syntax floats above all that, focusing on business logic like creating new records, fetching/searching existing records, updating records, or destroying records. No matter what database you're contacting, the usage is *exactly the same*. Furthermore, Waterline allows you to `.populate()` associations between models, *even if* the data for each model lives in a different database. That means you can switch your app's models from Mongo, to Postgres, to MySQL, to Redis, and back again - without changing any code. For the times when you need low-level, database-specific functionality, Waterline provides a query interface that allows you to talk directly to your models' underlying database driver (see .query() and .native().)

Scenario

Let's imagine you're building an e-commerce website, with an accompanying mobile app. Users browse products by category or search for products by keyword, then they buy them. That's it! Some parts of your app are quite ordinary; you have an API-driven flow for logging in, signing up, order/payment processing, resetting passwords, etc. However, you know there are a few mundane features lurking in your roadmap that will likely become more involved. Sure enough:

Flexibility

You ask the business what database they would like to use:

> "Datab... what? Let's not be hasty, wouldn't want to make the wrong choice. I'll get ops/IT on it. Go ahead and get started though."

The traditional methodology of choosing one single database for a web application/API is actually prohibitive for many production use cases. Oftentimes the application needs to maintain compatibility with one or more existing data sets, or it is necessary to use a few different types of databases for performance reasons.

Since Sails uses `sails-disk` by default, you can start building your app with zero configuration, using a local temporary file as storage. When you're ready to switch to the real thing (and when everyone knows what that even is), just change your app's connection configuration.

Compatibility

The product owner/stakeholder walks up to you and says:

> "Oh hey by the way, the products actually already live in our point of sale system. It's some ERP thing I guess, something like "DB2"? Anyways, I'm sure you'll figure it out- sounds easy right?"

Many enterprise applications must integrate with an existing database. If you're lucky, a one-time data migration may be all that's necessary, but more commonly, the existing dataset is still being modified by other applications. In order to build your app, you might need to marry data from multiple legacy systems, or with a separate dataset stored elsewhere. These datasets could live on 5 different servers scattered across the world! One colocated database server might house a SQL database with relational data, while another cloud server might hold a handful of Mongo or Redis collections.

Sails/Waterline lets you hook up different models to different datastores; locally or anywhere on the internet. You can build a User model that maps to a custom MySQL table in a legacy database (with weird crazy column names). Same thing for a Product model that maps to a table in DB2, or an Order model that maps to a MongoDB collection. Best of all, you can `.populate()` across these different datastores and adapters, so if you configure a model to live in a different database, your controller/model code doesn't need to change (note that you *will* need to migrate any important production data manually)

Performance

You're sitting in front of your laptop late at night, and you realize:

> "How can I do keyword search? The product data doesn't have any keywords, and the business wants search results ranked based on n-gram word sequences. Also I have no idea how this recommendation engine is going to work. Also if I hear the words `big data` one more time tonight I'm quitting and going back to work at the coffee shop."

So what about the "big data"? Normally when you hear bloggers and analyst use that buzzword, you think of data mining and business intelligence. You might imagine a process that pulls data from multiple sources, processes/indexes/analyzes it, then writes that extracted information somewhere else and either keeps the original data or throws it away.

However, there are some much more common challenges that lend themselves to the same sort of indexing/analysis needs; features like "driving-direction-closeness" search, or a recommendation engine for related products. Fortunately, a number of databases simplify specific big-data use cases (for instance MongoDB provides geospatial indexing, and ElasticSearch provides excellent support for indexing data for full-text search).

Using databases in the way they're intended affords tremendous performance benefits, particularly when it comes to complex report queries, searching (which is really just customized sorting), and NLP/machine learning. Certain databases are very good at answering traditional relational business queries, while others are better suited for map/reduce-style processing of data, with optimizations/trade-offs for blazing-fast read/writes. This consideration is especially important as your app's user-base scales.

Adapters

Like most MVC frameworks, Sails supports multiple databases. That means the syntax to query and manipulate our data is always the same, whether we're using MongoDB, MySQL, or any other supported database.

Waterline builds on this flexibility with its concept of adapters. An adapter is a bit of code that maps methods like `find()` and `create()` to a lower-level syntax like `SELECT * FROM` and `INSERT INTO`. The Sails core team maintains open-source adapters for a handful of the most popular databases, and a wealth of community adapters are also available.

Custom Waterline adapters are actually pretty simple to build, and can make for more maintainable integrations; anything from a proprietary enterprise system, to an open API like LinkedIn, to a cache or traditional database.

Connections

A **connection** represents a particular database configuration. This configuration object includes an adapter to use, as well as information like the host, port, username, password, and so forth. If your database doesn't require a password simply delete the password

property. Connections are defined in `config/connections.js` .

```
// in config/connections.js
// ...
{
  adapter: 'sails-mysql',
  host: 'localhost',
  port: 3306,
  user: 'root',
  password: 'g3tInCr4zee&stUfF'
}
// ...
```

The default database connection for a Sails app is located in the base model configuration (`config/models.js`), but it can also be overriden on a per-model basis by specifying a `connection` .

Analogy

Imagine a file cabinet full of completed pen-and-ink forms. All of the forms have the same fields (e.g. "name", "birthdate", "maritalStatus"), but for each form, the *values* written in the fields vary. For example, one form might contain "Lara", "2000-03-16T21:16:15.127Z", "single", while another form contains "Larry", "1974-01-16T21:16:15.127Z", "married".

Now imagine you're running a hotdog business. If you were *very* organized, you might set up your file cabinets as follows:

- **Employee** (contains your employee records)
  - `fullName`
  - `hourlyWage`
  - `phoneNumber`
- **Location** (contains a record for each location you operate)
  - `streetAddress`
  - `city`
  - `state`
  - `zipcode`
  - `purchases`
    - a list of all the purchases made at this location
  - `manager`
    - the employee who manages this location
- **Purchase** (contains a record for each purchase made by one of your customers)
  - `madeAtLocation`
  - `productsPurchased`

  - `createdAt`
- **Product** (contains a record for each of your various product offerings)
  - `nameOnMenu`
  - `price`
  - `numCalories`
  - `percentRealMeat`
  - `availableAt`
    - a list of the locations where this product offering is available.

In your Sails app, a **model** is like one of the file cabinets. It contains **records**, which are like the forms. `Attributes` are like the fields in each form.

Notes

- This documentation on models is not applicable if you are overriding the built-in ORM, Waterline. In that case, your models will follow whatever convention you set up, on top of whatever ORM library you're using (e.g. Mongoose.)

Associations

With Sails and Waterline, you can associate models across multiple data stores. This means that even if your users live in PostgreSQL and their photos live in MongoDB, you can interact with the data as if they lived together in the same database. You can also have associations that span different connections (i.e. datastores/databases) using the same adapter. This comes in handy if, for example, your app needs to access/update legacy recipe data stored in a MySQL database in your company's data center, but also store/retrieve ingredient data from a brand new MySQL database in the cloud.

Dominance

Example Ontology

```
// User.js
module.exports = {
  connection: 'ourMySQL',
  attributes: {
    email: 'string',
    wishlist: {
      collection: 'product',
      via: 'wishlistedBy'
    }
  }
};
```

```
// Product.js
module.exports = {
  connection: 'ourRedis',
  attributes: {
    name: 'string',
    wishlistedBy: {
      collection: 'user',
      via: 'wishlist'
    }
  }
};
```

The Problem

It's easy to see what's going on in this cross-adapter relationship. There's a many-to-many (`N->...`) relationship between users and products. In fact, you can imagine a few other relationships (e.g. purchases) which might exist, but since those are probably better-represented using a middleman model, I went for something simple in this example.

Anyways, that's all great... but where does the relationship resource live? "ProductUser", if you'll pardon with the SQL-oriented nomenclature. We know it'll end up on one side or the other, but what if we want to control which database it ends up in?

> **IMPORTANT NOTE**
>
> This is *only a problem because both sides of the association have a* `via` *modifier specified*!! In the absence of `via`, a collection attribute always behaves as `dominant: true`. See the FAQ below for more information.

The Solution

Eventually, it may even be possible to specify a 3rd connection/adapter to use for the join table. For now, we'll focus on choosing one side or the other.

We address this through the concept of "dominance." In any cross-adapter model relationship, one side is assumed to be dominant. It may be helpful to think about the analogy of a child with multinational parents who must choose one country or the other for her citizenship

Here's the ontology again, but this time we'll indicate the MySQL database as the "dominant". This means that the "ProductUser" relationship "table" will be stored as a MySQL table.

```
// User.js
module.exports = {
  connection: 'ourMySQL',
  attributes: {
    email: 'string',
    wishlist: {
      collection: 'product',
      via: 'wishlistedBy',
      dominant: true
    }
  }
};
```

```
// Product.js
module.exports = {
  connection: 'ourRedis',
  attributes: {
    name: 'string',
    wishlistedBy: {
      collection: 'user',
      via: 'wishlist'
    }
  }
};
```

Choosing a "dominant"

Several factors may influence your decision:

- If one side is a SQL database, placing the relationship table on that side will allow your queries to be more efficient, since the relationship table can be joined before the other side is communicated with. This reduces the number of total queries required from 3 to 2.
- If one connection is much faster than the other, all other things being equal, it probably makes sense to put the connection on that side.
- If you know that it is much easier to migrate one of the connections, you may choose to set that side as `dominant`. Similarly, regulations or compliance issues may affect your decision as well. If the relationship contains sensitive patient information (for instance, a relationship between `Patient` and `Medicine`) you want to be sure that all relevant data is saved in one particular database over the other (in this case, `Patient` is likely to be `dominant`).
- Along the same lines, if one of your connections is read-only (perhaps `Medicine` in the previous example is connected to a read-only vendor database), you won't be able to write to it, so you'll want to make sure your relationship data can be persisted safely on the other side.

FAQ

What if one of the collections doesn't have `via`?

> If a `collection` association does not have a `via` property, it is automatically `dominant: true`.

What if both collections don't have `via`?

> If both `collections` don't have `via`, then they are not related. Both are `dominant`, because they are separate relationship tables!!

What about `model` associations?

> In all other types of associations, the `dominant` property is prohibited. Setting one side to `dominant` is only necessary for associations between two models which have an attribute like: `{ via: '...', collection: '...' }` on both sides.

Can a model be dominant for one attribute and not another?

> Keep in mind that a model is "dominant" only in the context of a particular relationship. A model may be dominant in one or more relationships (attributes) while simultaneously NOT being dominant in other relationships (attributes). e.g. if a `User` has a collection of toys called `favoriteToys` via `favoriteToyOf` on the `Toy` model, and `favoriteToys` on `User` is `dominant: true`, `Toy` can still be dominant in other ways. So `Toy` might also be associated to `User` by way of its attribute, `designedBy`, for which it is `dominant: true`.

Can both models be dominant?

> No. If both models in a cross-adapter/cross-connection, many-to-many association set `dominant: true`, an error is thrown before lift.

Can neither model be dominant?

> Sort of... If neither model in a cross-adapter/cross-connection, many-to-many association sets `dominant: true`, a warning is displayed before lift, and a guess will be made automatically based on the characteristics of the relationship. For now, that just means an arbitrary decision based on alphabetical order :)

What about non-cross-adapter associations?

> The `dominant` property is silently ignored in non-cross-adapter/cross-connection associations. We're assuming you might be planning on breaking up the schema across multiple connections eventually, and there's no reason to prevent you from being proactive. Plus, this reserves additional future utility for the "dominant" option down the road.

Many-to-Many

Overview

A many-to-many association states that a model can be associated with many other models and vice-versa. Because both models can have many related models a new join table will need to be created to keep track of these relations.

Waterline will look at your models and if it finds that two models both have collection attributes that point to each other, it will automatically build up a join table for you.

Because you may want a model to have multiple many-to-many associations on another model a `via` key is needed on the `collection` attribute. This states which `model` attribute on the one side of the association is used to populate the records.

Using the `User` and `Pet` example lets look at how to build a schema where a `User` may have many `Pet` records and a `Pet` may have multiple owners.

Many-to-Many Example

In this example, we will start with an array of users and an array of pets. We will create records for each element in each array then associate all of the `Pets` with all of the `Users`. If everything worked properly, we should be able to query any `User` and see that they 'own' all of the `Pets`. Furthermore, we should be able to query any `Pet` and see that it is 'owned' by every `User`.

`myApp/api/models/pet.js`

```
module.exports = {

    attributes: {
        name:'STRING',
        color:'STRING',

        // Add a reference to User
        owners: {
            collection: 'user',
            via: 'pets'
        }
    }
}
```

`myApp/api/models/user.js`

```
module.exports = {

    attributes: {
        name:'STRING',
        age:'INTEGER',

        // Add a reference to Pet
        pets:{
            collection: 'pet',
            via: 'owners'
        }
    }

}
```

`myApp/config/bootstrap.js`

```
module.exports.bootstrap = function (cb) {

// After we create our users, we will store them here to associate with our pets
var storeUsers = [];

var users = [{name:'Mike',age:'16'},{name:'Cody',age:'25'},{name:'Gabe',age:'107'}];
var ponys = [{ name: 'Pinkie Pie', color: 'pink'},{ name: 'Rainbow Dash',color: 'blue'},{

// This does the actual associating.
// It takes one Pet then iterates through the array of newly created Users, adding each o
var associate = function(onePony,cb){
  var thisPony = onePony;
  var callback = cb;

  storeUsers.forEach(function(thisUser,index){
    console.log('Associating ',thisPony.name,'with',thisUser.name);
    thisUser.pets.add(thisPony.id);
    thisUser.save(console.log);

    if (index === storeUsers.length-1)
      return callback(thisPony.name);
  })
};


// This callback is run after all of the Pets are created.
// It sends each new pet to 'associate' with our Users
var afterPony = function(err,newPonys){
  while (newPonys.length){
    var thisPony = newPonys.pop();
    var callback = function(ponyID){
      console.log('Done with pony ',ponyID)
```

```
    }
    associate(thisPony,callback)
  }
  console.log('Everyone belongs to everyone!! Exiting.');

  // This callback lets us leave bootstrap.js and continue lifting our app!
  return cb()
};

// This callback is run after all of our Users are created.
// It takes the returned User and stores it in our storeUsers array for later.
var afterUser = function(err,newUsers){
  while (newUsers.length)
    storeUsers.push(newUsers.pop())

  Pet.create(ponys).exec(afterPony)
};


User.create(users).exec(afterUser)

};
```

Lifting our app with `sails console`

```
dude@littleDude:~/node/myApp$ sails console

info: Starting app in interactive mode...

Associating  Applejack with Gabe
Associating  Applejack with Cody
Associating  Applejack with Mike
Done with pony  Applejack
Associating  Rainbow Dash with Gabe
Associating  Rainbow Dash with Cody
Associating  Rainbow Dash with Mike
Done with pony  Rainbow Dash
Associating  Pinkie Pie with Gabe
Associating  Pinkie Pie with Cody
Associating  Pinkie Pie with Mike
Done with pony  Pinkie Pie
Everyone belongs to everyone!! Exiting.
info: Welcome to the Sails console.
info: ( to exit, type <CTRL>+<C> )

sails> null { name: 'Gabe',
  age: 107,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 9 }
```

```
null { name: 'Cody',
  age: 25,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 8 }
null { name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
null { name: 'Gabe',
  age: 107,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 9 }
null { name: 'Cody',
  age: 25,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 8 }
null { name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
null { name: 'Gabe',
  age: 107,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 9 }
null { name: 'Cody',
  age: 25,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 8 }
null { name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
sails> Pet.find().populate('owners').exec(function(e,r){while(r.length){var thisPet=r.pop
{ owners:
   [ { name: 'Mike',
       age: 16,
       id: 7,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Cody',
       age: 25,
       id: 8,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Gabe',
```

```
        age: 107,
        id: 9,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
  name: 'Applejack',
  color: 'orange',
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 9 }
{ owners:
   [ { name: 'Mike',
       age: 16,
       id: 7,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Cody',
       age: 25,
       id: 8,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Gabe',
       age: 107,
       id: 9,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
  name: 'Rainbow Dash',
  color: 'blue',
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 8 }
{ owners:
   [ { name: 'Mike',
       age: 16,
       id: 7,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Cody',
       age: 25,
       id: 8,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Gabe',
       age: 107,
       id: 9,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
  name: 'Pinkie Pie',
  color: 'pink',
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
sails> User.find().populate('pets').exec(function(e,r){while(r.length){var thisUser=r.pop
{ pets:
```

```
    [ { name: 'Pinkie Pie',
        color: 'pink',
        id: 7,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Rainbow Dash',
        color: 'blue',
        id: 8,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Applejack',
        color: 'orange',
        id: 9,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
   name: 'Gabe',
   age: 107,
   createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
   updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
   id: 9 }
 { pets:
    [ { name: 'Pinkie Pie',
        color: 'pink',
        id: 7,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Rainbow Dash',
        color: 'blue',
        id: 8,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Applejack',
        color: 'orange',
        id: 9,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
   name: 'Cody',
   age: 25,
   createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
   updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
   id: 8 }
 { pets:
    [ { name: 'Pinkie Pie',
        color: 'pink',
        id: 7,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Rainbow Dash',
        color: 'blue',
        id: 8,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Applejack',
```

```
        color: 'orange',
        id: 9,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
    name: 'Mike',
    age: 16,
    createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
    updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
    id: 7 }
```

Notes

> For a more detailed description of this type of association, see the Waterline Docs

One Way Association

Overview

A one way association is where a model is associated with another model. You could query that model and populate to get the associated model. You can't however query the associated model and populate to get the associating model.

One Way Example

In this example, we are associating a `User` with a `Pet` but not a `Pet` with a `User`.

`myApp/api/models/pet.js`

```
module.exports = {

    attributes: {
        name:'STRING',
        color:'STRING'
    }

}
```

`myApp/api/models/user.js`

```
module.exports = {

    attributes: {
        name:'STRING',
        age:'INTEGER',
        pony:{
            model: 'pet'
        }
    }

}
```

Using `sails console`

```
sails> Pet.create({name:'Pinkie Pie',color:'pink'}).exec(console.log)
null { name: 'Pinkie Pie',
  color: 'pink',
  createdAt: Tue Feb 11 2014 15:45:33 GMT-0600 (CST),
  updatedAt: Tue Feb 11 2014 15:45:33 GMT-0600 (CST),
  id: 5 }

sails> User.create({name:'Mike',age:21,pony:5}).exec(console.log);
null { name: 'Mike',
  age: 21,
  pony: 5,
  createdAt: Tue Feb 11 2014 15:48:53 GMT-0600 (CST),
  updatedAt: Tue Feb 11 2014 15:48:53 GMT-0600 (CST),
  id: 1 }

sails> User.find({name:'Mike'}).populate('pony').exec(console.log);
null [ { name: 'Mike',
    age: 21,
    pony:
     { name: 'Pinkie Pie',
       color: 'pink',
       id: 5,
       createdAt: Tue Feb 11 2014 15:45:33 GMT-0600 (CST),
       updatedAt: Tue Feb 11 2014 15:45:33 GMT-0600 (CST) },
    createdAt: Tue Feb 11 2014 15:48:53 GMT-0600 (CST),
    updatedAt: Tue Feb 11 2014 15:48:53 GMT-0600 (CST),
    id: 1 } ]
```

Notes

> For a more detailed description of this type of association, see the Waterline Docs
>
> Because we have only formed an association on one of the models, a `Pet` has no restrictions on the number of `User` models it can belong to. If we wanted to, we could change this and associate the `Pet` with exactly one `User` and the `User` with exactly one `Pet`.

One-to-Many

Overview

A one-to-many association states that a model can be associated with many other models. To build this association a virtual attribute is added to a model using the `collection` property. In a one-to-many association one side must have a `collection` attribute and the other side must contain a `model` attribute. This allows the many side to know which records it needs to get when a `populate` is used.

Because you may want a model to have multiple one-to-many associations on another model a `via` key is needed on the `collection` attribute. This states which `model` attribute on the one side of the association is used to populate the records.

One-to-Many Example

`myApp/api/models/pet.js`

```
module.exports = {

    attributes: {
        name:'STRING',
        color:'STRING',
        owner:{
            model:'user'
        }
    }

}
```

`myApp/api/models/user.js`

```
module.exports = {

    attributes: {
        name:'STRING',
        age:'INTEGER',
        pets:{
            collection: 'pet',
            via: 'owner'
        }
    }

}
```

Using `sails console`

```
sails> User.create({name:'Mike',age:'21'}).exec(console.log)
null { pets: [Getter/Setter],
  name: 'Mike',
  age: 21,
  createdAt: Tue Feb 11 2014 17:49:04 GMT-0600 (CST),
  updatedAt: Tue Feb 11 2014 17:49:04 GMT-0600 (CST),
  id: 1 }

sails> Pet.create({name:'Pinkie Pie',color:'pink',owner:1}).exec(console.log)
null { name: 'Pinkie Pie',
    color: 'pink',
    owner: 1,
    createdAt: Tue Feb 11 2014 17:58:04 GMT-0600 (CST),
    updatedAt: Tue Feb 11 2014 17:58:04 GMT-0600 (CST),
    id: 2 }

sails> Pet.create({name:'Applejack',color:'orange',owner:1}).exec(console.log)
null { name: 'Applejack',
    color: 'orange',
    owner: 1,
    createdAt: Tue Feb 11 2014 18:02:58 GMT-0600 (CST),
    updatedAt: Tue Feb 11 2014 18:02:58 GMT-0600 (CST),
    id: 4 }

sails> User.find().populate('pets').exec(function(err,r){console.log(r[0].toJSON())});
{ pets:
   [ { name: 'Pinkie Pie',
       color: 'pink',
       id: 2,
       createdAt: Tue Feb 11 2014 17:58:04 GMT-0600 (CST),
       updatedAt: Tue Feb 11 2014 17:58:04 GMT-0600 (CST),
       owner: 1 },
     { name: 'Applejack',
       color: 'orange',
```

```
        id: 4,
        createdAt: Tue Feb 11 2014 18:02:58 GMT-0600 (CST),
        updatedAt: Tue Feb 11 2014 18:02:58 GMT-0600 (CST),
        owner: 1 } ],
    name: 'Mike',
    age: 21,
    createdAt: Tue Feb 11 2014 17:49:04 GMT-0600 (CST),
    updatedAt: Tue Feb 11 2014 17:49:04 GMT-0600 (CST),
    id: 1 }

  sails> Pet.find(4).populate('owner').exec(console.log)
  null [ { name: 'Applejack',
      color: 'orange',
      owner:
       { pets: [Getter/Setter],
         name: 'Mike',
         age: 21,
         id: 1,
         createdAt: Tue Feb 11 2014 17:49:04 GMT-0600 (CST),
         updatedAt: Tue Feb 11 2014 17:49:04 GMT-0600 (CST) },
      createdAt: Tue Feb 11 2014 18:02:58 GMT-0600 (CST),
      updatedAt: Tue Feb 11 2014 18:02:58 GMT-0600 (CST),
      id: 4 } ]
```

Notes

> For a more detailed description of this type of association, see the Waterline Docs

One-to-One

Overview

A one-to-one association states that a model may only be associated with one other model. In order for the model to know which other model it is associated with, a foreign key must be included in the record.

One-to-One Example

In this example, we are associating a `Pet` with a `User`. The `User` may only have one `Pet` and viceversa, a `Pet` can only have one `User`. However, in order to query this association from both sides, you will have to create/update both models.

`myApp/api/models/pet.js`

```
module.exports = {

    attributes: {
        name:'STRING',
        color:'STRING',
        owner:{
            model:'user'
        }
    }

}
```

`myApp/api/models/user.js`

```
module.exports = {

    attributes: {
        name:'STRING',
        age:'INTEGER',
        pony:{
            model: 'pet'
        }
    }

}
```

Using `sails console`

```
sails> User.create({ name: 'Mike', age: 21}).exec(console.log);
null { name: 'Mike',
  age: 21,
  createdAt: Thu Feb 20 2014 17:12:18 GMT-0600 (CST),
  updatedAt: Thu Feb 20 2014 17:12:18 GMT-0600 (CST),
  id: 1 }

sails> Pet.create({ name: 'Pinkie Pie', color: 'pink', owner: 1}).exec(console.log)
null { name: 'Pinkie Pie',
    color: 'pink',
    owner: 1,
    createdAt: Thu Feb 20 2014 17:26:16 GMT-0600 (CST),
    updatedAt: Thu Feb 20 2014 17:26:16 GMT-0600 (CST),
    id: 2 }

sails> Pet.find().populate('owner').exec(console.log)
null [ { name: 'Pinkie Pie',
    color: 'pink',
    owner:
     { name: 'Mike',
       age: 21,
       id: 1,
       createdAt: Thu Feb 20 2014 17:12:18 GMT-0600 (CST),
       updatedAt: Thu Feb 20 2014 17:12:18 GMT-0600 (CST) },
    createdAt: Thu Feb 20 2014 17:26:16 GMT-0600 (CST),
    updatedAt: Thu Feb 20 2014 17:26:16 GMT-0600 (CST),
    id: 2 } ]

sails> User.find().populate('pony').exec(console.log)
null [ { name: 'Mike',
    age: 21,
    createdAt: Thu Feb 20 2014 18:11:15 GMT-0600 (CST),
    updatedAt: Thu Feb 20 2014 18:11:15 GMT-0600 (CST),
    id: 2,
    pony: undefined } ]

sails> User.update({name:'Mike'},{pony:2}).exec(console.log)
null [ { name: 'Mike',
    age: 21,
    createdAt: Thu Feb 20 2014 17:12:18 GMT-0600 (CST),
    updatedAt: Thu Feb 20 2014 17:30:58 GMT-0600 (CST),
    id: 1,
    pony: 2 } ]

sails> User.findOne(1).populate('pony').exec(console.log)
null { name: 'Mike',
  age: 21,
  createdAt: Thu Feb 20 2014 17:12:18 GMT-0600 (CST),
  updatedAt: Thu Feb 20 2014 17:30:58 GMT-0600 (CST),
```

```
  id: 1,
  pony:
   { name: 'Pinkie Pie',
     color: 'pink',
     id: 2,
     createdAt: Thu Feb 20 2014 17:26:16 GMT-0600 (CST),
     updatedAt: Thu Feb 20 2014 17:26:16 GMT-0600 (CST),
     owner: 1 } }
```

Notes

> For a more detailed description of this type of association, see the Waterline Docs

Through Associations

Overview

Many-to-Many through associations behave the same way as many-to-many associations with the exception of the join table being automatically created for you. This allows you to attach additional attributes onto the relationship inside of the join table.

Unfortunately, they are not supported yet. Don't worry though, there's an easy workaround.

You can accomplish this by using an additional model as an intermediary. Instead of a many-to-many association between two models, you can use multiple one-to-many associations through the intermediary model.

Attributes

Overview

Model attributes are basic pieces of information about a model. A model called `Person` might have attributes called `firstName`, `lastName`, `phoneNumber`, `age`, `birthDate` and `emailAddress`.

Attribute Options

These options can be used to enforce various constraints and add special enhancements to our model attributes.

type

Specifies the type of data that will be stored in this attribute. One of:

- string
- text
- integer
- float
- date
- datetime
- boolean
- binary
- array
- json

email

Checks the incoming value for a valid email address.

```
attributes: {
  email: {
    type: 'string',
    email: true
  }
}
```

defaultsTo

When a record is created, if no value was supplied, the record will be created with the specified `defaultsTo` value.

```
attributes: {
  phoneNumber: {
    type: 'string',
    defaultsTo: '111-222-3333'
  }
}
```

autoIncrement

Sets up the attribute as an auto-increment key. When a new record is added to the model, if a value for this attribute is not specified, it will be generated by incrementing the most recent record's value by one. Note: Attributes which specify `autoIncrement` should always be of `type: integer`. Also, bear in mind that the level of support varies across different datastores. For instance, MySQL will not allow more than one auto-incrementing column per table.

```
attributes: {
  placeInLine: {
    type: 'integer',
    autoIncrement: true
  }
}
```

unique

Ensures no two records will be allowed with the same value for the target attribute. This is an adapter-level constraint, so in most cases this will result in a unique index on the attribute being created in the underlying datastore.

```
attributes: {
  username: {
    type: 'string',
    unique: true
  }
}
```

primaryKey

Use this attribute as the the primary key for the record. Only one attribute per model can be the `primaryKey`. Note: This should never be used unless autoPK is set to false.

```
attributes: {
  uuid: {
    type: 'string',
    primaryKey: true,
    required: true
  }
}
```

enum

A special validation property which only saves data which matches a whitelisted set of values.

```
attributes: {
  state: {
    type: 'string',
    enum: ['pending', 'approved', 'denied']
  }
}
```

size

If supported in the adapter, can be used to define the size of the attribute. For example in MySQL, `size` can be specified as a number (`n`) to create a column with the SQL data type: `varchar(n)` .

```
attributes: {
  name: {
    type: 'string',
    size: 24
  }
}
```

columnName

Inside an attribute definition, you can specify a `columnName` to force Sails/Waterline to store data for that attribute in a specific column in the configured connection (i.e. database). Be aware that this is not necessarily SQL-specific-- it will also work for MongoDB fields, etc.

While the `columnName` property is primarily designed for working with existing/legacy databases, it can also be useful in situations where your database is being shared by other applications, or you don't have access permissions to change the schema.

To store/fetch your model's `numberOfWheels` attribute into/from the `number_of_round_rotating_things` column:

```
// An attribute in one of your models:
// ...
numberOfWheels: {
  type: 'integer',
  columnName: 'number_of_round_rotating_things'
}
// ...
```

Now for a more thorough/realistic example.

Let's say you have a `User` model in your Sails app that looks like this:

```
// api/models/User.js
module.exports = {
  connection: 'shinyNewMySQLDatabase',
  attributes: {
    name: 'string',
    password: 'string',
    email: {
      type: 'email',
      unique: true
    }
  }
};
```

Everything works great, but instead of using an existing MySQL database sitting on a server somewhere that happens to house your app's intended users:

```
// config/connections.js
module.exports = {
  // ...

  // Existing users are in here!
  rustyOldMySQLDatabase: {
    adapter: 'sails-mysql',
    user: 'bofh',
    host: 'db.eleven.sameness.foo',
    password: 'Gh19R!?had9gzQ#Q#Q#%AdsghaDABAMR>##G<ADMBOVRH@)$(HTOADG!GNADSGADSGNBI@(',
    database: 'jonas'
  },
  // ...
};
```

Let's say there's a table called `our_users` in the old MySQL database that looks like this:

the_primary_key	email_address	full_name	seriously_encrypted_passv
7	mike@sameness.foo	Mike McNeil	ranchdressing
14	nick@sameness.foo	Nick Crumrine	thousandisland

In order to use this from Sails, you'd change your `User` model to look like this:

```
// api/models/User.js
module.exports = {
  connection: 'rustyOldMySQLDatabase',
  tableName: 'our_users',
  attributes: {
    id: {
      type: 'integer',
      unique: true,
      primaryKey: true,
      columnName: 'the_primary_key'
    },
    name: {
      type: 'string',
      columnName: 'full_name'
    },
    password: {
      type: 'string',
      columnName: 'seriously_encrypted_password'
    },
    email: {
      type: 'email',
      unique: true,
      columnName: 'email_address'
    }
  }
};
```

> You might have noticed that we also used the `tableName` property in this example. This allows us to control the name of the table that will be used to house our data.

Lifecycle callbacks

Overview

Lifecycle callbacks are functions that are automagically called before or after certain *model* actions. For example, we sometimes use lifecycle callbacks to automatically encrypting a password before creating or updating an `Account` model.

Sails exposes a handful of lifecycle callbacks by default.

Callbacks on `create`

- beforeValidate: fn(values, cb)
- afterValidate: fn(values, cb)
- beforeCreate: fn(values, cb)
- afterCreate: fn(newlyInsertedRecord, cb)

Callbacks on `update`

- beforeValidate: fn(valuesToUpdate, cb)
- afterValidate: fn(valuesToUpdate, cb)
- beforeUpdate: fn(valuesToUpdate, cb)
- afterUpdate: fn(updatedRecord, cb)

Callbacks on `destroy`

- beforeDestroy: fn(criteria, cb)
- afterDestroy: fn(destroyedRecords, cb)

Example

If you want to encrypt a password before saving in the database, you might use the `beforeCreate` lifecycle callback.

```
var bcrypt = require('bcrypt');

module.exports = {

  attributes: {

    username: {
      type: 'string',
      required: true
    },

    password: {
      type: 'string',
      minLength: 6,
      required: true,
      columnName: 'encrypted_password'
    }

  },


  // Lifecycle Callbacks
  beforeCreate: function (values, cb) {

    // Encrypt password
    bcrypt.hash(values.password, 10, function(err, hash) {
      if(err) return cb(err);
      values.password = hash;
      //calling cb() with an argument returns an error. Useful for canceling the entire o
      cb();
    });
  }
};
```

Models

A model represents a collection of structured data, usually corresponding to a single table or collection in a database. Models are usually defined by creating a file in an app's `api/models/` folder.

```
// Parrot.js
// The set of parrots registered in our app.
module.exports = {
  attributes: {
    // e.g., "Polly"
    name: {
      type: 'string'
    },

    // e.g., 3.26
    wingspan: {
      type: 'float',
      required: true
    },

    // e.g., "cm"
    wingspanUnits: {
      type: 'string',
      enum: ['cm', 'in', 'm', 'mm'],
      defaultsTo: 'cm'
    },

    // e.g., [{...}, {...}, ...]
    knownDialects: {
      collection: 'Dialect'
    }
  }
}
```

Using models

Models may be accessed from our controllers, policies, services, responses, tests, and in custom model methods. There are many built-in methods available on models, the most important of which are the query methods: find, create, update, and destroy. These methods are asynchronous - under the covers, Waterline has to send a query to the database and wait for a response.

Consequently, query methods return a deferred query object. To actually execute a query, `.exec(cb)` must be called on this deferred object, where `cb` is a callback function to run after the query is complete.

Waterline also includes opt-in support for promises. Instead of calling `.exec()` on a query object, we can call `.then()`, `.spread()`, or `.catch()`, which will return a Bluebird promise.

Model Methods (aka "static" or "class" methods)

Model class methods are functions built into the model itself that perform a particular task on its instances (records). This is where you will find the familiar CRUD methods for performing database operations like `.create()`, `.update()`, `.destroy()`, `.find()`, etc.

Custom model methods

Waterline allows you to define custom methods on your models. This feature takes advantage of the fact that Waterline models ignore unrecognized keys, so you do need to be careful about inadvertently overriding built-in methods and dynamic finders (don't define methods named "create", etc.) Custom model methods are most useful for extrapolating controller code that relates to a particular model; i.e. this allows you to pull code out of your controllers and into reusuable functions that can be called from anywhere (i.e. don't depend on `req` or `res`.)

Model methods are generally asynchronous functions. By convention, asynchronous model methods should be 2-ary functions, which accept an object of inputs as their first argument (usually called `opts` or `options`) and a Node callback as the second argument. Alternatively, you might opt to return a promise (both strategies work just fine- it's a matter of preference. If you don't have a preference, stick with Node callbacks.)

A best practice is to write your static model method so that it can accept either a record OR its primary key value. For model methods that operate on/from *multiple* records at once, you should allow an array of records OR an array of primary key values to be passed in. This takes more time to write, but makes your method much more powerful. And since you're doing this to extrapolate commonly-used logic anyway, it's usually worth the extra effort.

For example:

```
// in api/models/Monkey.js...

// Find monkeys with the same name as the specified person
findWithSameNameAsPerson: function (opts, cb) {

  var person = opts.person;

  // Before doing anything else, check if a primary key value
  // was passed in instead of a record, and if so, lookup which
  // person we're even talking about:
  (function _lookupPersonIfNecessary(afterLookup){
    // (this self-calling function is just for concise-ness)
    if (typeof person === 'object')) return afterLookup(null, person);
    Person.findOne(person).exec(afterLookup);
  })(function (err, person){
    if (err) return cb(err);
    if (!person) {
      err = new Error();
      err.message = require('util').format('Cannot find monkeys with the same name as the
      err.status = 404;
      return cb(err);
    }

    Monkey.findByName(person.name)
    .exec(function (err, monkeys){
      if (err) return cb(err);
      cb(null, monkeys);
    })
  });

}
```

Then you can do:

```
Monkey.findWithSameNameAsPerson(albus, function (err, monkeys) { ... });
// -or-
Monkey.findWithSameNameAsPerson(37, function (err, monkeys) { ... });
```

> For more tips, read about the incident involving Timothy the Monkey.

Another example:

```
// api/models/User.js
module.exports = {

  attributes: {

    name: {
      type: 'string'
    },
    enrolledIn: {
      collection: 'Course', via: 'students'
    }
  },

  /**
   * Enrolls a user in one or more courses.
   * @param  {Object}   options
   *             => courses {Array} list of course ids
   *             => id {Integer} id of the enrolling user
   * @param  {Function} cb
   */
  enroll: function (options, cb) {

    User.findOne(options.id).exec(function (err, theUser) {
      if (err) return cb(err);
      if (!theUser) return cb(new Error('User not found.'));
      theUser.enrolledIn.add(options.courses);
      theUser.save(cb);
    });
  }
};
```

Dynamic Finders

These are special static methods that are dynamically generated by Sails when you lift your app. For instance, if your Person model has a "firstName", you might run:

```
Person.findByFirstName('emma').exec(function(err,people){ ... });
```

Resourceful Pubsub Methods

A special type of model methods which are attached by the pubsub hook. More on that in the section of the docs on resourceful pubsub.

Attribute Methods (i.e. record/instance methods)

Attribute methods are functions available on records (i.e. model instances) returned from Waterline queries. For example, if you find the ten students with the highest GPA from the Student model, each of those student records will have access to all the built-in attribute methods, as well as any custom attribute methods defined on the Student model.

Built-in attribute methods

Every Waterline model includes some attribute methods automatically, including:

- `.toJSON()`
- `.save()`
- `.destroy()`
- `.validate()`

Custom attribute methods

Waterline models also allow you to define your own custom attribute methods. Define them like any other attribute, but instead of an attribute definition object, write a function on the right-hand-side.

```
// From api/models/Person.js...

module.exports = {
  attributes: {
    // Primitive attributes
    firstName: {
      type: 'string',
      defaultsTo: ''
    },
    lastName: {
      type: 'string',
      defaultsTo: ''
    },

    // Associations (aka relational attributes)
    spouse: { model: 'Person' },
    pets: { collection: 'Pet' },

    // Attribute methods
    getFullName: function (){
      return this.firstName + ' ' + this.lastName;
    },
    isMarried: function () {
      return !!this.spouse;
    },
    isEligibleForSocialSecurity: function (){
      return this.age >= 65;
    },
    encryptPassword: function () {

    }
  }
};
```

> Note that with the notable exception of the built-in `.save()` and `.destroy()` attribute methods, attribute methods are almost always *synchronous* by convention.

When to write a custom attribute method

Custom attribute methods are particularly useful for extracting some information out of a record. I.e. you might want to reduce some information from one or more attributes (i.e. "is this person married?"")

```
if ( rick.isMarried() ) {
  // ...
}
```

When NOT to write a custom attribute method

You should **avoid writing your own *asynchronous* attribute methods**. While built-in asynchronous attribute methods like `.save()` and `.destroy()` can be convenient from your app code, writing your *own* asynchronous attribute methods can sometimes have unintended consequences, and is not the most efficient way to build your app.

For instance, consider an app that manages wedding records. You might think to write an attribute method on the Person model that updates the `spouse` attribute on both individuals in the database. This would allow you to write controller code like:

```
personA.marry(personB, function (err) {
  if (err) return res.negotiate(err);
  return res.ok();
})
```

Which looks great...until you need to write a different action where you don't have an actual record for "personA".

A better strategy is to write a custom (static) model method instead. This makes your function more reusable/versatile, since it will be accessible whether or not you have an actual record instance on hand. You might refactor the code from the previous example to look like:

```
Person.marry([joe,raquel], function (err) {
  if (err) return res.negotiate(err);
  return res.ok();
})
```

Naming your attribute methods

Make sure you use a naming convention that helps you avoid confusing **attribute methods** from *attribute values* when you're working with records in your app. A good best practice is to use "get*" (e.g. `getFullName()`) prefix and avoid writing attribute methods that change records in-place.

Waterline Query Language

The Waterline Query language is an object-based criteria used to retrieve the records from any of the supported database adapters. This means that you can use the same query on MySQL as you do on Redis or MongoDb. This allows you to change your database without changing your code.

Query Language Basics

The criteria objects are formed using one of four types of object keys. These are the top level keys used in a query object. It is loosely based on the criteria used in MongoDB with a few slight variations.

Queries can be built using either a `where` key to specify attributes, which will allow you to also use query options such as `limit` and `skip` or if `where` is excluded the entire object will be treated as a `where` criteria.

```
Model.find({ where: { name: 'foo' }, skip: 20, limit: 10, sort: 'name DESC' });

// OR

Model.find({ name: 'foo' })
```

Key Pairs

A key pair can be used to search records for values matching exactly what is specified. This is the base of a criteria object where the key represents an attribute on a model and the value is a strict equality check of the records for matching values.

```
Model.find({ name: 'walter' })
```

They can be used together to search multiple attributes.

```
Model.find({ name: 'walter', state: 'new mexico' })
```

Modified Pairs

Modified pairs also have model attributes for keys but they also use any of the supported criteria modifiers to perform queries where a strict equality check wouldn't work.

```
Model.find({
  name : {
    'contains' : 'alt'
  }
})
```

In Pairs

IN queries work similarly to mysql 'in queries'. Each element in the array is treated as 'or'.

```
Model.find({
  name : ['Walter', 'Skyler']
});
```

Not-In Pairs

Not-In queries work similar to `in` queries, except for the nested object criteria.

```
Model.find({
  name: { '!' : ['Walter', 'Skyler'] }
});
```

Or Pairs

Performing `OR` queries is done by using an array of query pairs. Results will be returned that match any of the criteria objects inside the array.

```
Model.find({
  or : [
    { name: 'walter' },
    { occupation: 'teacher' }
  ]
})
```

Criteria Modifiers

The following modifiers are available to use when building queries.

- `'<'` / `'lessThan'`
- `'<='` / `'lessThanOrEqual'`
- `'>'` / `'greaterThan'`
- `'>='` / `'greaterThanOrEqual'`

- `'!'` / `'not'`
- `'like'`
- `'contains'`
- `'startsWith'`
- `'endsWith'`

'<' / 'lessThan'

Searches for records where the value is less than the value specified.

```
Model.find({ age: { '<': 30 }})
```

'<=' / 'lessThanOrEqual'

Searches for records where the value is less or equal to the value specified.

```
Model.find({ age: { '<=': 21 }})
```

'>' / 'greaterThan'

Searches for records where the value is more than the value specified.

```
Model.find({ age: { '>': 18 }})
```

'>=' / 'greaterThanOrEqual'

Searches for records where the value is more or equal to the value specified.

```
Model.find({ age: { '>=': 21 }})
```

'!' / 'not'

Searches for records where the value is not equal to the value specified.

```
Model.find({ name: { '!': 'foo' }})
```

'like'

Searches for records using pattern matching with the `%` sign. (Case insensitive.)

```
Model.find({ food: { 'like': '%beans' }})
```

'contains'

A shorthand for pattern matching both sides of a string. Will return records where the value contains the string anywhere inside of it. (Case insensitive.)

```
Model.find({ class: { 'contains': 'history' }})

// The same as

Model.find({ class: { 'like': '%history%' }})
```

'startsWith'

A shorthand for pattern matching the right side of a string. Will return records where the value starts with the supplied string value. (Case insensitive.)

```
Model.find({ class: { 'startsWith': 'american' }})

// The same as

Model.find({ class: { 'like': 'american%' }})
```

'endsWith'

A shorthand for pattern matching the left side of a string. Will return records where the value ends with the supplied string value. (Case insensitive.)

```
Model.find({ class: { 'endsWith': 'can' }})

// The same as

Model.find({ class: { 'like': '%can' }})
```

'Date Ranges'

You can do date range queries using the comparison operators.

```
Model.find({ date: { '>': new Date('2/4/2014'), '<': new Date('2/7/2014') } })
```

Query Options

Query options allow you refine the results that are returned from a query. The current options available are:

- `limit`
- `skip`
- `sort`

Limit

Limits the number of results returned from a query.

```
Model.find({ where: { name: 'foo' }, limit: 20 })
```

Skip

Returns all the results excluding the number of items to skip.

```
Model.find({ where: { name: 'foo' }, skip: 10 });
```

Pagination

`skip` and `limit` can be used together to build up a pagination system.

```
Model.find({ where: { name: 'foo' }, limit: 10, skip: 10 });
```

`paginate` is a Waterline helper method which can accomplish the same as `skip` and `limit`.

```
Model.find().paginate({page: 2, limit: 10});
```

> **Waterline**
>
> You can find out more about the Waterline API below:
>
> - Sails.js Documentation
> - Waterline README
> - Waterline Documentation
> - Waterline Github Repository

Sort

Results can be sorted by attribute name. Simply specify an attribute name for natural (ascending) sort, or specify an `asc` or `desc` flag for ascending or descending orders respectively.

```
// Sort by name in ascending order
Model.find({ where: { name: 'foo' }, sort: 'name' });

// Sort by name in descending order
Model.find({ where: { name: 'foo' }, sort: 'name DESC' });

// Sort by name in ascending order
Model.find({ where: { name: 'foo' }, sort: 'name ASC' });

// Sort by binary notation
Model.find({ where: { name: 'foo' }, sort: { 'name': 1 }});

// Sort by multiple attributes
Model.find({ where: { name: 'foo' }, sort: { name:  1, age: 0 });
```

> **Case-sensitivity**
>
> All queries inside of Waterline are **case-insensitive**. This allows for easier querying but makes indexing strings tough. This is something to be aware of if you are indexing and searching on string fields.
>
> Currently, the best way to execute **case-sensitive** queries is using the `.native()` or `.query()` method.

Validations

Sails bundles support for automatic validations of your models' attributes. Any time a record is updated, or a new record is created, the data for each attribute will be checked against all of your predefined validation rules. This provides a convenient failsafe to ensure that invalid entries don't make their way into your app's database(s).

Validation Rules

Validations are handled by Anchor, a thin layer on top of Validator, one of the most robust validation libraries for Node.js. Sails supports most of the validations available in Validator, as well as a few extras that require database integration, like `unique`.

Name of validator	What does it check?	Notes on usage
after	check if `string` date in this record is after the specified `Date`	must be valid javascript `Date`
alpha	check if `string` in this record contains only letters (a-zA-Z)	
alphadashed		does this `string` contain only letters and/or dashes?
alphanumeric	check if `string` in this record contains only letters and numbers.	
alphanumericdashed	does this `string` contain only numbers and/or letters and/or dashes?	
array	is this a valid javascript `array` object?	strings formatted as arrays won't pass
before	check if `string` in this record is a date that's before the specified date	
binary	is this binary data?	If it's a string, it will always pass
boolean	is this a valid javascript `boolean` ?	`string` s will fail
contains	check if `string` in this record contains the seed	
creditcard	check if `string` in this record is a credit card	

date	check if `string` in this record is a date	takes both strings and javascript
datetime	check if `string` in this record looks like a javascript `datetime`	
decimal		contains a decimal or is less than 1?
email	check if `string` in this record looks like an email address	
empty	Arrays, strings, or arguments objects with a length of 0 and objects with no own enumerable properties are considered "empty"	lo-dash _.isEmpty()
equals	check if `string` in this record is equal to the specified value	`===` ! They must match in both value and type
falsey	Would a Javascript engine register a value of `false` on this?	
finite	Checks if given value is, or can be coerced to, a finite number	This is not the same as native isFinite which will return true for booleans and empty strings
float	check if `string` in this record is of the number type float	
hexadecimal	check if `string` in this record is a hexadecimal number	
hexColor	check if `string` in this record is a hexadecimal color	
in	check if `string` in this record is in the specified array of allowed `string` values	
int	check if `string` in this record is an integer	
integer	same as above	Im not sure why there are two of these.
ip	check if `string` in this record is a valid IP (v4 or v6)	
ipv4	check if `string` in this record is a valid IP v4	
ipv6	check if `string` in this record is a valid IP v6	
is		something to do with REGEX

json	does a try&catch to check for valid JSON.	
len	is `integer` > param1 && < param2	Where are params defined?
lowercase	is this string in all lowercase?	
max		
maxLength	is `integer` > 0 && < param2	
min		
minLength		
not		Something about regexes
notContains		
notEmpty		
notIn	does the value of this model attribute exist inside of the defined validator value (of the same type)	Takes strings and arrays
notNull	does this not have a value of `null` ?	
notRegex		
null	check if `string` in this record is null	
number	is this a number?	NaN is considered a number
numeric	checks if `string` in this record contains only numbers	
object	checks if this attribute is the language type of Object	Passes for arrays, functions, objects, regexes, new Number(0), and new String('') !
regex		
protected	Should this attribute be removed when `toJSON` is called on a model instance?	
required	Must this model attribute contain valid data before a new record can be created?	
string	is this a `string` ?	
text		

text	okay, well is *this* a `string` ?	
truthy	Would a Javascript engine register a value of `false` on this?	
undefined	Would a javascript engine register this thing as have the value 'undefined' ?	
unique	Checks to see if a new record model attribute is unique.	
uppercase	checks if `string` in this record is uppercase	
url	checks if `string` in this record is a URL	
urlish	Does the `string` in this record contain something that looks like a route, ending with a file extension?	/^\s([\/]+.)+.+\s*$/g
uuid	checks if `string` in this record is a UUID (v3, v4, or v5)	
uuidv3	checks if `string` in this record is a UUID (v3)	
uuidv4	checks if `string` in this record is a UUID (v4)	

Custom Validation Rules

You can define your own types and their validation with the types object. It's possible to access and compare values to other attributes (with "this"). This allows you to move validation business logic into your models and out of your controller logic.

> Note that your own type always have to be a variant of the basic data-types ("string", "int", "json", ...)

Example Model

```
// api/models/foo
module.exports = {

  types: {
    is_point: function(geoLocation){
      return geoLocation.x && geoLocation.y
    },
    password: function(password) {
      return password === this.passwordConfirmation;
    }
  },
  attributes: {
    firstName: {
      type: 'string',
      required: true,
      minLength: 5,
      maxLength: 15
    },
    location: {
      //note, that the base type (json) still has to be defined
      type: 'json',
      is_point: true
    },
    password: {
      type: 'string',
      password: true
    },
    passwordConfirmation: {
      type: 'string'
    }

  }
}
```

Custom Validation Messages

Out of the box, Sails.js does not support custom validation messages. However, for Sails v0.11.0+ a Hook is available: sails-hook-validator. Details regarding its usage can be found in the sails-hook-validator repository.

Model Settings

The following properties can be specified at the top level of your model definition to override the defaults for that particular model. To override the default settings shared by all of your models, edit `config/models.js` .

`migrate`

```
migrate: 'safe'
```

In short, this setting controls whether/how Sails will attempt to automatically rebuild the tables/collections/sets/etc. in your schema.

In a production environment (NODE_ENV==="production") Sails always uses `migrate:"safe"` to protect inadvertent deletion of your data. However during development, you have a few other options for convenience:

1. safe - never auto-migrate my database(s). I will do it myself (by hand)
2. alter - auto-migrate, but attempt to keep my existing data (experimental)
3. drop - wipe/drop ALL my data and rebuild models every time I lift Sails

When your sails app lifts, waterline validates all of the data in your database. This flag tells waterline what to do with data when the data is corrupt. You can set this flag to `safe` which will ignore the corrupt data and continue to lift. You can also set it to

Auto-Migration Strategy	Description
`safe`	never auto-migrate my database(s). I will do it myself, by hand.
`alter`	auto-migrate columns/fields, but attempt to keep my existing data (experimental)
`drop`	wipe/drop ALL my data and rebuild models every time I lift Sails

> Note, by using `drop` , or even `alter` , you risk losing your data. Be careful. Never use `drop` or `alter` with a production dataset. Additionally, on large databases `alter` may take a long time to complete at startup. This may cause commands like `sails console` to appear to hang.

`schema`

```
schema: true
```

A flag to toggle schemaless or schema mode in databases that support schemaless data structures. If turned off, this will allow you to store arbitrary data in a record. If turned on, only attributes defined in the model's `attributes` object will be stored.

For adapters that don't require a schema, such as Mongo or Redis, the default setting is `schema:false`.

`connection`

```
connection: 'my-local-postgresql'
```

The configured database connection where this model will fetch and save its data. Defaults to `localDiskDb`, the default connection that uses the `sails-disk` adapter.

`identity`

```
identity: 'purchase'
```

The lowercase unique key for this model, e.g. `user`. By default, a model's `identity` is inferred automatically by lowercasing its filename. You should never change this property on your models.

`globalId`

```
globalId: 'Purchase'
```

This flag changes the global name by which you can access your model (if the globalization of models is enabled). You should never change this property on your models. To disable globals, see `sails.config.globals`.

autoPK

```
autoPK: true
```

A flag to toggle the automatic definition of a primary key in your model. The details of this default PK vary between adapters (e.g. MySQL uses an auto-incrementing integer primary key, whereas MongoDB uses a randomized string UUID). In any case, the primary keys generated by autoPK will be unique. If turned off no primary key will be created by default, and you will need to define one manually, e.g.:

```
attributes: {
  sku: {
    type: 'string',
    primaryKey: true,
    unique: true
  }
}
```

`autoCreatedAt`

```
autoCreatedAt: true
```

A flag to toggle the automatic definition of a `createdAt` attribute in your model. By default, `createdAt` is an attribute which will be automatically set when a record is created with the current timestamp, e.g.:

```
attributes: {
  createdAt: {
    type: 'datetime',
    defaultsTo: function (){ return new Date(); }
  }
}
```

`autoUpdatedAt`

```
autoUpdatedAt: true
```

A flag to toggle the automatic definition of a `updatedAt` attribute in your model. By default, `updatedAt` is an attribute which will be automatically set with the current timestamp every time a record is updated, e.g.:

```
attributes: {
  updatedAt: {
    type: 'datetime',
    defaultsTo: function (){ return new Date(); }
  }
}
```

tableName

```
tableName: 'some_preexisting_table'
```

You can define a custom name for the physical collection in your adapter by adding a `tableName` attribute. **This isn't just for tables**. In MySQL, PostgreSQL, Oracle, etc. this setting refers to the name of the table, but in MongoDB or Redis, it refers to the collection, and so forth. If no tableName is specified, Waterline will use the model's `identity` as its `tableName`.

This is particularly useful for working with pre-existing/legacy databases.

`attributes`

```
attributes: {
  name: { type: 'string' },
  email: { type: 'email' },
  age: { type: 'integer' }
}
```

See Attributes.

Policies

Overview

Policies in Sails are versatile tools for authorization and access control-- they let you allow or deny access to your controllers down to a fine level of granularity. For example, if you were building Dropbox, before letting a user upload a file to a folder, you might check that she `isAuthenticated` , then ensure that she `canWrite` (has write permissions on the folder.) Finally, you'd want to check that the folder she's uploading into `hasEnoughSpace` .

Policies can be used for anything: HTTP BasicAuth, 3rd party single-sign-on, OAuth 2.0, or your own custom authorization/authentication scheme.

> NOTE: policies apply **only** to controller actions, not to views. If you define a route in your routes.js config file that points directly to a view, no policies will be applied to it. To make sure policies are applied, you can instead define a controller action which displays your view, and point your route to that action.

Writing Your First Policy

Policies are files defined in the `api/policies` folder in your Sails app. Each policy file should contain a single function.

When it comes down to it, policies are really just Connect/Express middleware functions which run **before** your controllers. You can chain as many of them together as you like-- in fact they're designed to be used this way. Ideally, each middleware function should really check just *one thing*.

For example, the `canWrite` policy mentioned above might look something like this:

```
// policies/canWrite.js
module.exports = function canWrite (req, res, next) {
  var targetFolderId = req.param('id');
  var userId = req.session.user.id;

  Permission
  .findOneByFolderId( targetFolderId )
  .exec( function foundPermission (err, permission) {

    // Unexpected error occurred-- skip to the app's default error (500) handler
    if (err) return next(err);

    // No permission exists linking this user to this folder.  Maybe they got removed fro
    if ( ! permission ) return res.redirect('/notAllowed');

    // OK, so a permission was found.  Let's be sure it's a "write".
    if ( permission.type !== 'write' ) return res.redirect('/notAllowed');

    // If we made it all the way down here, looks like everything's ok, so we'll let the
    next();
  });
};
```

Protecting Controllers with Policies

Sails has a built in ACL (access control list) located in `config/policies.js`. This file is used to map policies to your controllers.

This file is *declarative*, meaning it describes *what* the permissions for your app should look like, not *how* they should work. This makes it easier for new developers to jump in and understand what's going on, plus it makes your app more flexible as your requirements inevitably change over time.

Your `config/policies.js` file should export a Javascript object whose keys are controller names (or `'*'` for global policies), and whose values are objects mapping action names to one or more policies. See below for more details and examples.

To apply a policy to a specific controller action:

```
{
  ProfileController: {
      // Apply the 'isLoggedIn' policy to the 'edit' action of 'ProfileController'
      edit: 'isLoggedIn'
      // Apply the 'isAdmin' AND 'isLoggedIn' policies, in that order, to the 'create' ac
      create: ['isAdmin', 'isLoggedIn']
  }
}
```

To apply a policy to an entire controller:

```
{
  ProfileController: {
    // Apply 'isLoggedIn' by default to all actions that are NOT specified below
    '*': 'isLoggedIn',
    // If an action is explicitly listed, its policy list will override the default list.
    // So, we have to list 'isLoggedIn' again for the 'edit' action if we want it to be a
    edit: ['isAdmin', 'isLoggedIn']
  }
}
```

> **Note:** Default policy mappings do not "cascade" or "trickle down." Specified mappings for the controller's actions will override the default mapping.

To apply a policy to all actions that are not explicitly mapped:

```
{
  // Apply 'isLoggedIn' to all actions by default
  '*': 'isLoggedIn',
  ProfileController: {
      // Apply 'isAdmin' to the 'foo' action.  'isLoggedIn' will NOT be applied!
      'foo': 'isAdmin'
  }
}
```

> Remember, default policies will not be applied to any controller / action that is given an explicit mapping.

Built-in policies

Sails provides two built-in policies that can be applied globally, or to a specific controller or action.

- `true` : public access (allows anyone to get to the mapped controller/action)

- `false` : **NO** access (allows **no-one** to access the mapped controller/action)

  `'*': true` is the default policy for all controllers and actions. In production, it's good practice to set this to `false` to prevent access to any logic you might have inadvertently exposed.

Adding some policies to a controller:

```
// in config/policies.js

// ...
RabbitController: {

  // Apply the `false` policy as the default for all of RabbitController's actions
  // (`false` prevents all access, which ensures that nothing bad happens to our rabbit
  '*': false,

  // For the action `nurture`, apply the 'isRabbitMother' policy
  // (this overrides `false` above)
  nurture : 'isRabbitMother',

  // Apply the `isNiceToAnimals` AND `hasRabbitFood` policies
  // before letting any users feed our rabbits
  feed : ['isNiceToAnimals', 'hasRabbitFood']
}
// ...
```

Here’s what the `isNiceToAnimals` policy from above might look like (this file would be located at `policies/isNiceToAnimals.js`):

```
module.exports = function isNiceToAnimals (req, res, next) {

  // `req.session` contains a set of data specific to the user making this request.
  // It's kind of like our app's "memory" of the current user.

  // If our user has a history of animal cruelty, not only will we
  // prevent her from going even one step further (`return`),
  // we'll go ahead and redirect her to PETA (`res.redirect`).
  if ( req.session.user.hasHistoryOfAnimalCruelty ) {
    return res.redirect('http://PETA.org');
  }

  // If the user has been seen frowning at puppies, we have to assume that
  // they might end up being mean to them, so we'll
  if ( req.session.user.frownsAtPuppies ) {
    return res.redirect('http://www.dailypuppy.com/');
  }

  // Finally, if the user has a clean record, we'll call the `next()` function
  // to let them through to the next policy or our controller
  next();
};
```

Besides protecting rabbits (while a noble cause, no doubt), here are a few other use cases for policies:

- cookie-based authentication
- role-based access control
- limiting file uploads based on MB quotas
- any other kind of authentication scheme you can imagine

Authentication and Permissioning with Sails + Passport

Passport works great with Sails! In general, since Sails uses Connect/Express at its core, all of the Connect/Express-oriented things work pretty well. In fact, Sails has no problem interpreting most Express middleware to work with socket.io.

Community-supported Sails extensions using passport.js

- sails-auth: Passport-based Authentication Extension, including Basic Auth
- sails-permissions: Permissions and Entitlements system for sails.js: supports user authentication with passport.js, role-based permissioning, object ownership, and row-level security.
- sails-generate-auth: Generate a Passport.js authentication layer for your Sails app
- Tutorial on how to implement passport.js with sails.js.
- Waterlock: An all encompassing user authentication/json web token management tool, built for Sails

Routes

Overview

The most basic feature of any web application is the ability to interpret a request sent to a URL, then send back a response. In order to do this, your application has to be able to distinguish one URL from another.

Like most web frameworks, Sails provides a router: a mechanism for mapping URLs to controllers and views. **Routes** are rules that tell Sails what to do when faced with an incoming request. There are two main types of routes in Sails: **custom** (or "explicit") and **automatic** (or "implicit").

Custom Routes

Sails lets you design your app's URLs in any way you like- there are no framework restrictions.

Every Sails project comes with `config/routes.js`, a simple Node.js module that exports an object of custom, or "explicit" **routes**. For example, this `routes.js` file defines six routes; some of them point to a controller's action, while others route directly to a view.

```
// config/routes.js
module.exports.routes = {
  'get /signup': { view: 'conversion/signup' },
  'post /signup': 'AuthController.processSignup',
  'get /login': { view: 'portal/login' },
  'post /login': 'AuthController.processLogin',
  '/logout': 'AuthController.logout',
  'get /me': 'UserController.profile'
}
```

Each **route** consists of an **address** (on the left, e.g. `'get /me'`) and a **target** (on the right, e.g. `'UserController.profile'`) The **address** is a URL path and (optionally) a specific HTTP method. The **target** can be defined a number of different ways (see the expanded concepts section on the subject), but the two different syntaxes above are the most common. When Sails receives an incoming request, it checks the **address** of all custom routes for matches. If a matching route is found, the request is then passed to its **target**.

For example, we might read `'get /me': 'UserController.profile'` as:

> "Hey Sails, when you receive a GET request to `http://mydomain.com/me`, run the `profile` action of `UserController`, would'ya?"

What if I want to change the view layout within the route itself? No problem we could:

```
'get /privacy': {
    view: 'users/privacy',
    locals: {
      layout: 'users'
    }
  },
```

Notes

- Just because a request matches a route address doesn't necessarily mean it will be passed to that route's target *directly*. For instance, HTTP requests will usually pass through some middleware first. And if the route points to a controller action, the request will need to pass through any configured policies first. Finally, there are a few special route options which allow a route to be "skipped" for certain kinds of requests.
- The router can also programmatically **bind** a **route** to any valid route target, including canonical Node middleware functions (i.e. `function (req, res, next) {}`). However, you should always use the conventional route target syntax when possible- it streamlines development, simplifies training, and makes your app more maintainable.

Automatic Routes

In addition to your custom routes, Sails binds many routes for you automatically. If a URL doesn't match a custom route, it may match one of the automatic routes and still generate a response. The main types of automatic routes in Sails are:

- Blueprint routes, which provide your controllers and models with a full REST API.
- Assets, such as images, Javascript and stylesheet files.
- CSRF, if turned on, provides a **/csrfToken** route to your app that can be used to retrieve the CSRF token.

Supported Protocols

The Sails router is "protocol-agnostic"; it knows how to handle both HTTP requests and messages sent via WebSockets. It accomplishes this by listening for Socket.io messages sent to reserved event handlers in a simple format, called JWR (JSON-WebSocket Request/Response). This specification is implemented and available out of the box in the client-side socket SDK.

Notes

- Advanced users may opt to circumvent the router entirely and send low-level, completely customizable WebSocket messages directly to the underlying Socket.io server. You can bind socket events directly in your app's `onConnect` function (located in `config/sockets.js`.) But bear in mind that, in most cases, you are better off leveraging the request interpreter for socket communication - maintaining consistent routes across HTTP and WebSockets helps keep your app maintainable.

Custom Routes

Overview

Sails allows you to explicitly route URLs in several different ways in your **config/routes.js** file. Every route configuration consists of an **address** and a **target**, for example:

```
'GET /foo/bar': 'FooController.bar'
^^^address^^^^  ^^^^^^target^^^^^^^
```

Route Address

The route address indicates what URL should be matched in order to apply the handler and options defined by the target. A route consists of an optional verb and a mandatory path:

```
'POST  /foo/bar'
^verb^ ^^path^^
```

If no verb is specified, the target will be applied to any request that matches the path, regardless of the HTTP method used (**GET**, **POST**, **PUT** etc.). Note the initial `/` in the path--all paths should start with one in order to work properly.

Wildcards and dynamic parameters

In addition to specifying a static path like **foo/bar**, you can use `*` as a wildcard:

```
'/*'
```

will match all paths, where as:

```
'/user/foo/*'
```

will match all paths that *start* with **/user/foo**.

> **Note:** When using a route with a wildcard, such as `'/*'`, be aware that this will also match requests to static assets (i.e. `/js/dependencies/sails.io.js`) and override them. To prevent this, consider using the `skipAssets` option described below.

You can capture the parts of the address that are matched by wildcards into named parameters by using the `:paramName` wildcard syntax instead of the `*` :

```
'/user/foo/:name/bar/:age'
```

Will match the same URLs as:

```
'/user/foo/*/bar/*'
```

but will provide the values of the wildcard portions of the route to the route handler as `req.param('name')` and `req.param('age')` , respectively.

Regular expressions in addresses

In addition to the wildcard address syntax, you may also use Regular Expressions to define the URLs that a route should match. The syntax for defining an address with a regular expression is:

```
"r|<regular expression string>|<comma-delimited list of param names>"
```

That's the letter "**r**", followed by a pipe character `|` , a regular expression string *without delimiters*, another pipe, and a list of parameter names that should be mapped to parenthesized groups in the regular expression. For example:

```
"r|^/\\d+/(\\w+)/(\\w+)$|foo,bar": "MessageController.myaction"
```

Will match `/123/abc/def` , running the `myaction` action of `MessageController` and supplying the values `abc` and `def` as `req.param('foo')` and `req.param('bar')` , respectively.

Note the double-backslash in `\\d` and `\\w` ; this escaping is necessary for the regular expression to work correctly!

About route ordering

When using wildcards or regular expressions in your addresses, keep in mind that the ordering of your routes in **config/routes.js** matters; URLs are matched against addresses in the list from the top down. If you have two configurations in this order:

```
'/user': 'UserController.doSomething',
'/*'   : 'CatchallController.doSomethingElse'
```

then a request to `/user` will not be matched by the second configuration unless the first configuration's handler calls `next()` in its code, which is discouraged (only policies should call `next()`). Unless you're doing something very advanced, it is safe to assume that every request will be handled by at most one route in your **config/routes.js** file.

Route Target

The address portion of a custom route specifies which URLs the route should match. The *target* portion specifies what Sails should do after the match is made. A target can take one of several different forms. In some cases you may want to chain multiple targets to a single address by placing them in an array, but in most cases each address will have only one target. The different types of targets are discussed below, followed by a discussion of the various options that can be applied to them.

Controller / action target syntax

The most common type of target is one which binds a route to a custom controller action. The following four routes are equivalent:

```
'GET /foo/go': 'FooController.myGoAction',
'GET /foo/go': 'Foo.myGoAction',
'GET /foo/go': {controller: "Foo", action: "myGoAction"},
'GET /foo/go': {controller: "FooController", action:"myGoAction"},
```

Each one maps `GET /foo/go` to the `myGoAction` action of the controller in **api/controllers/FooController.js**. If no such controller or action exists, Sails will output an error message and ignore the route. Otherwise, whenever a **GET** request to **/foo/go** is made, the code in that action will be run.

The controller and action names in this syntax are case-insensitive.

Note that the blueprints API adds several actions to your controllers by default (like "find", "create", "update" and "delete"), all of which are available for routing:

```
'GET /foo/go': 'UserController.find'
```

Assuming that your have a **api/controllers/UserController.js** file and a **api/models/User.js** file, browsing to **/foo/go** in a browser will, using the above config, run the default "find* blueprint action which displays a list of all `User` models. If you have a custom action named `find` in UserController, that action will be run instead.

View target syntax

Another common target is one that binds a route to a view. The syntax for this is simple: it's the path to the view file relative to the **views** folder, without the file extension:

```
'GET /home': {view: 'home/index'}
```

This maps the `GET /home` to the view stored in **views/home/index.ejs** (assuming the default EJS template engine is used). As long as that view file exists, a **GET** request to **/home** will display it.

> Note that this route will be bound directly to the view, no policies will be applied in this case. See the StackOverflow question for more information.

Blueprint target syntax

In some cases you may want to map a non-standard address to one of the Sails blueprint actions. For example, if you have a controller and model named **UserController** and **User** respectively, then Sails will automatically map **GET /user** to the "find" blueprint action which returns a list of User records. If you'd like to map a different address to that action, you can use the blueprint target syntax:

```
'GET /findAllUsers': {model: 'user', blueprint: 'find'},
'GET /user/findAll': {blueprint: 'find'}
```

Note that in the configuration, both the `model` and `blueprint` properties are set, while in the second one, only `blueprint` is used. In the second configuration, leaving off the `model` property causes Sails to examine the address and guess that the model is `User`. You could override this by explicitly setting `model` to something else:

```
'GET /user/findAll': {blueprint: 'find', model: 'pet'}
```

although you will rarely if ever want to do this, as it makes for a messy and confusing API for your app.

If you specify a non-existent model or blueprint in your configuration, Sails will output an error and ignore the route.

You can also use this syntax to map a route to one of the default blueprint actions even if you've overridden that action in a controller.

Redirect target syntax

You can have one address redirect to another--either within your Sails app, or on another server entirely--you can do so just by specifying the redirect URL as a string:

```
'/alias' : '/some/other/route'
'GET /google': 'http://www.google.com'
```

Be careful to avoid redirect loops when redirecting within your Sails app!

Note that when redirecting, the HTTP method of the original request (and any extra headers / parameters) will likely be lost, and the request will be transformed to a simple **GET** request. In the above example, a **POST** request to **/alias** will result in a **GET** request to **/some/other/route**. This is somewhat browser-dependent behavior, but it is recommended that you don't expect request methods and other data to survive a redirect.

Response target syntax

You can map an address directly to a default or custom response using this syntax:

```
'/foo': {response: 'notFound'}
```

Simply specify the name of the response file in your **api/responses** folder, without the **.js** extension. The response name in this syntax is case-sensitive. If you attempt to bind a route to a non-existent response, Sails will output an error and ignore the route.

Policy target syntax

In most cases, you will want to apply policies to your controller actions using the **config/policies.js** config file. However, there are some times when you will want to apply a policy directly to a custom route: particularly when you are using the view or blueprint target syntax. The policy target syntax is:

```
'/foo': {policy: 'myPolicy'}
```

However, you will always want to chain the policy to at least one other type of target, using an array:

```
'/foo': [{policy: 'myPolicy'}, {blueprint: 'find', model: 'user'}]
```

This will apply the **myPolicy** policy to the route and, if it passes, continue by running the **find** blueprint for the **User** model.

Route target options

In addition to the options discussed in the various route target syntaxes above, any other property you add to a route target object will be passed through to the route handler in the `req.options` object. There are several reserved properties that can be used to affect the behavior of the route handlers. These are listed in the table below.

Property	Applicable Target Types	Data Type	Details
`skipAssets`	all	((boolean))	Set to `true` if you *don't* want the route to match URLs with dots in them (e.g. **myImage.jpg**). This will keep your routes with wildcard notation from matching URLs of static assets. Useful when creating URL slugs.
`skipRegex`	all	((regexp))	If skipping every URL containing a dot is too permissive, or you need a route's handler to be skipped based on different criteria entirely, you can use `skipRegex`. This option allows you to specify a regular expression or array of regular expressions to match the request URL against; if any of the matches are successful, the handler is skipped. Note that unlike the syntax for binding a handler with a regular expression, `skipRegex` expects *actual RegExp objects*, not strings.
`locals`	controller, view, blueprint, response	((object))	Sets default local variables to pass to any view that is rendered while handling the request.
`cors`	all	((object)) or ((boolean)) or ((string))	Specifies how to handle requests for this route from a different origin. See the main CORS documentation for more info.
`populate`	blueprint	((boolean))	Indicates whether the results in a "find" or "findOne" blueprint action should have associated model fields populated. Defaults to the value set in **config/blueprints.js**.
`skip`, `limit`, `sort`, `where`	blueprint	((object))	Set criteria for "find" blueprint. See the queries reference for more info.

URL Slugs

A common use case for explicit routes is the design of slugs or vanity URLs. For example, consider the URL of a repository on Github, `http://www.github.com/balderdashy/sailsjs`. In Sails, we might define this route at the **bottom of our `config/routes.js` file** like so:

```
'get /:account/:repo': {
controller: 'RepoController',
action: 'show',
skipAssets: true
}
```

In your `RepoController`'s `show` action, we'd use `req.param('account')` and `req.param('repo')` to look up the data for the appropriate repository, then pass it in to the appropriate view as locals. The `skipAssets` option ensures that the vanity route doesn't accidentally match any of our assets (e.g. `/images/logo.png`), so they are still accessible.

Security

Overview

Sails and Express provide built-in, easily configurable protection against most known types of web-application-level attacks.

> **Note**: If you believe you have found a security vulnerability in Sails, please refer to our security policy for instructions for reporting it.

Cross-Origin Resource Sharing (CORS)

CORS is a mechanism that allows browser scripts on pages served from other domains (e.g. myothersite.com) to talk to your server (e.g. api.mysite.com). Like JSONP, the goal of CORS is to function as a secure method to circumvent the same-origin policy; allowing your Sails server to successfully respond to requests from client-side JavaScript code running on a page from some other domain. But unlike JSONP, it works with more than just GET requests.

Sails can be configured to allow cross-origin requests from a list of domains you specify, or from every domain. This can be done on a per-route basis, or globally for every route in your app.

Enabling CORS

For security reasons, CORS is disabled by default in Sails. But enabling it is dead-simple.

To allow cross-origin requests from *any* domain to *any* route in your app, simply enable `allRoutes` in `config/cors.js` :

```
allRoutes: true
```

See `sails.config.cors` for a comprehensive reference of all available options.

Configuring CORS For Individual Routes

Besides the global CORS configuration, you can set up individual routes in `config/routes.js` to accept (or deny) cross-origin requests. To indicate that a route should accept CORS requests using the configuration parameters in `config/cors.js` , set its `cors` property to `true` :

```
"get /foo": {
   controller: "FooController",
   action: "index",
   cors: true
}
```

If you have the `allRoutes` parameter set to `true` in `config.cors.js` , but you want to exempt a specific route, you can do so by explicitly setting its `cors` property to `false` :

```
"get /foo": {
   controller: "FooController",
   action: "index",
   cors: false
}
```

To override specific CORS configuration parameters for a route, add a `cors` property object:

```
"get /foo": {
   controller: "FooController",
   action: "index",
   cors: {
     origin: "http://sailsjs.org, http://sailsjs.com",
     credentials: false
   }
}
```

Security Levels

By default, Sails will still process all the requests that come in regardless of domain, even with CORS enabled: it will simply set the appropriate headers on the response so that the *client* can decide whether or not to show the response. For example, if you send a `GET` request to `/foo/bar` from a domain that is not in your CORS whitelist, the `bar` action in your `FooController.js` file will still run, but the browser will throw away the result. This may seem counterintuitive, but it is important because it allows non-browser-based clients (like Postman and curl) to work while still blocking the kind of attacks that the Same-Origin Policy is meant to protect against.

If you want to completely prevent Sails from processing requests from disallowed domains, you can use the `securityLevel` setting:

```
module.exports.cors = {
  allRoutes: true,
  origin: "http://sailsjs.org",
  securityLevel: 1
}
```

Security level 1 (high) will respond with a 403 status code to any request from a disallowed origin prefixed with the `http` or `https` protocol. Security level 2 (very high) will do the same, but extended to *all* protocols (so things like Postman and curl won't work).

CSRF

Cross-site request forgery (CSRF)) is a type of attack which forces an end user to execute unwanted actions on a web application backend with which he/she is currently authenticated. In other words, without protection, cookies stored in a browser like Google Chrome can be used to send requests to Chase.com from a user's computer whether that user is currently visiting Chase.com or Horrible-Hacker-Site.com.

Enabling CSRF Protection

Sails bundles optional CSRF protection out of the box. To enable the built-in enforcement, just make the following adjustment to sails.config.csrf (conventionally located in your project's `config/csrf.js` file):

```
csrf: true
```

Note that if you have existing code that communicates with your Sails backend via POST, PUT, or DELETE requests, you'll need to acquire a CSRF token and include it as a parameter or header in those requests. More on that in a sec.

CSRF Tokens

Like most Node applications, Sails and Express are compatibile with Connect's CSRF protection middleware for guarding against such attacks. This middleware implements the Synchronizer Token Pattern. When CSRF protection is enabled, all non-GET requests to the Sails server must be accompanied by a special token, identified by either a header or a parameter in the query string or HTTP body.

CSRF tokens are temporary and session-specific; e.g. Imagine Mary and Muhammad are both shoppers accessing our e-commerce site running on Sails, and CSRF protection is enabled. Let's say that on Monday, Mary and Muhammad both make purchases. In order to do so, our site needed to dispense at least two different CSRF tokens- one for Mary and one for Muhammad. From then on, if our web backend received a request with a missing or incorrect token, that request will be rejected. So now we can rest assured that when Mary navigates away to play online poker, the 3rd party website cannot trick the browser into sending malicious requests to our site using her cookies.

Dispensing CSRF Tokens

To get a CSRF token, you should either bootstrap it in your view using locals (good for traditional multi-page web applications) or fetch it using sockets or AJAX from a special protected JSON endpoint (handy for single-page-applications (SPAs).)

Using View Locals:

For old-school form submissions, it's as easy as passing the data from a view into a form action. You can grab hold of the token in your view, where it may be accessed as a view local: `<%= _csrf %>`

e.g.:

```
<form action="/signup" method="POST">
 <input type="text" name="emailaddress"/>
 <input type='hidden' name='_csrf' value='<%= _csrf %>'>
 <input type='submit'>
</form>
```

If you are doing a `multipart/form-data` upload with the form, be sure to place the `_csrf` field before the `file` input, otherwise you run the risk of a timeout and a 403 firing before the file finishes uploading.

Using AJAX/WebSockets

In AJAX/Socket-heavy apps, you might prefer to send a GET request to the built-in `/csrfToken` route, where it will be returned as JSON, e.g.:

```
{
  "_csrf": "ajg4JD(JGdajhLJALHDa"
}
```

Spending CSRF Tokens

Once you've enabled CSRF protection, any POST, PUT, or DELETE requests (**including** virtual requests, e.g. from Socket.io) made to your Sails app will need to send an accompanying CSRF token as a header or parameter. Otherwise, they'll be rejected with a 403 (Forbidden) response.

For example, if you're sending an AJAX request from a webpage with jQuery:

```
$.post('/checkout', {
  order: '8abfe13491afe',
  electronicReceiptOK: true,
  _csrf: 'USER_CSRF_TOKEN'
}, function andThen(){ ... });
```

With some client-side modules, you may not have access to the AJAX request itself. In this case, you can consider sending the CSRF token directly in the URL of your query. However, if you do so, remember to URL-encode the token before spending it:

```
..., {
  checkoutAction: '/checkout?_csrf='+encodeURIComponent('USER_CSRF_TOKEN')
}
```

Notes

> - For most developers and organizations, CSRF attacks need only be a concern if you allow users to log into/securely access your Sails backend from the browser. If you *don't* (e.g. users only access the secured sections from your native iOS or Android app), it is possible you don't need to enable CSRF protection. Why? Because technically, the common CSRF attack discussed on this page is only *possible* in scenarios where users use the *same client application* (e.g. Chrome) to access different web services (e.g. Chase.com, Horrible-Hacker-Site.com.)
> - For more information on CSRF, check out Wikipedia
> - For "spending" CSRF tokens in a traditional form submission, refer to the example above (under "Using View Locals".)
> - You can choose to send the CSRF token as a header instead of a parameter- refer to the Connect documentation for the most up-to-date information. The next (post v0.10) minor release of Sails will likely upgrade to Express 4, at which point the new Express csrf middleware will be included instead, but backwards compatibility will be maintained.

Clickjacking

Clickjacking (aka "UI redress attacks") are where an attacker manages to trick your users into triggering "unintended" UI events (e.g. DOM events.)

X-FRAME-OPTIONS

One simple way to help prevent clickjacking attacks is to enable the X-FRAME-OPTIONS header.

Using lusca

> `lusca` is open-source under the Apache license

```
# In your sails app
npm install lusca --save
```

Then in the `middleware` config object in `config/http.js` :

```
  // ...
  // maxAge ==> Number of seconds strict transport security will stay in effect.
  xframe: require('lusca').xframe('SAMEORIGIN'),
  // ...
  order: [
    // ...
    'xframe'
    // ...
  ]
```

Additional Resources

- Clickjacking (OWasp)

Content Security Policy

Content Security Policy (CSP) is a W3C specification for instructing the client browser as to which location and/or which type of resources are allowed to be loaded. This spec uses "directives" to define a loading behaviors for target resource types. Directives can be specified using HTTP response headers or or HTML Meta tags.

HTTP Headers

Header	Browsers
Content-Security-Policy	(W3C Standard header) Chrome version >= 25, Firefox version >= 23, Opera version >= 19
X-Content-Security-Policy	Firefox version < 23, IE version 10
X-WebKit-CSP	Chrome version < 25

Supported Directives

Directive	
default-src	Loading policy for all resources type in case a resource type dedicated directive is not defined (fallback)
script-src	Defines which scripts the protected resource can execute
object-src	Defines from where the protected resource can load plugins
style-src	Defines which styles (CSS) the user applies to the protected resource
img-src	Defines from where the protected resource can load images
media-src	Defines from where the protected resource can load video and audio
frame-src	Defines from where the protected resource can embed frames
font-src	Defines from where the protected resource can load fonts
connect-src	Defines which URIs the protected resource can load using script interfaces
form-action	Defines which URIs can be used as the action of HTML form elements
sandbox	Specifies an HTML sandbox policy that the user agent applies to the protected resource
script-nonce	Defines script execution by requiring the presence of the specified nonce on script elements
plugin-types	Defines the set of plugins that can be invoked by the protected resource by limiting the types of resources that can be embedded
reflected-xss	Instructs a user agent to activate or deactivate any heuristics used to filter or block reflected cross-site scripting attacks, equivalent to the effects of the non-standard X-XSS-Protection header
report-uri	Specifies a URI to which the user agent sends reports about policy violation

> For more information, see the W3C CSP Spec

DDOS

The prevention of denial of service attacks is a complex problem which involves multiple layers of protection, up and down the networking stack. This type of attack has achieved notoriety in recent years due to widespread media coverage of groups like Anonymous.

At the API layer, there isn't much that can be done in the way of prevention. However, Sails offers a few settings to mitigate certain types of DDOS attacks:

- The session in Sails can be configured to use a separate session store (e.g. Redis), allowing your application to run without relying on the memory state of any one API server. This means that multiple copies of your Sails app may be deployed to as many servers as is necessary to handle traffic. This is achieved by using a load balancer), which directs each incoming request to a free server with the resources to handle it, eliminating any one app server as a single point of failure.
- Socket.io connections may be configured to use a separate socket store (e.g. Redis) for managing pub/sub state and message queueing. This eliminates the need for sticky sessions at the load balancer, preventing would-be attackers from directing their attacks against the same server again and again.

Additional Resources

- Backpressure and Unbounded Concurrency in Node.js (Voxer)
- Building a Node.js Server That Won't Melt (Mozilla)
- Security in Node.js - see the "Denial of Service" section (Harry Torry)
- Slowloris DDoSAttacks

P3P

Background

P3P stands for the "Platform for Privacy Preferences", a browser/web standard designed to facilitate better consumer web privacy control. Currently (as of 2014), out of all the major browsers, it is only supported by Internet Explorer. It comes into play most often when dealing with legacy applications.

Many modern organizations are willfully ignoring P3P. Here's what Facebook has to say on the subject:

> The organization that established P3P, the World Wide Web Consortium, suspended its work on this standard several years ago because most modern web browsers don't fully support P3P. As a result, the P3P standard is now out of date and doesn't reflect technologies that are currently in use on the web, so most websites currently don't have P3P policies.
>
> See also: http://www.zdnet.com/blog/facebook/facebook-to-microsoft-p3p-is-outdated-what-else-ya-got/9332

Supporting P3P with Sails

But all that aside, sometimes you just have to support P3P anyways.

Fortunately, a few different modules exist that bring P3P support to Express and Sails by enabling the relevant P3P headers. To use one of these modules for handling P3P headers, install it from npm using the directions below, then open `config/http.js` in your project and configure it as a custom middleware. To do that, define your P3P middleware as "p3p", and add the string "p3p" to your `middleware.order` array wherever you'd like it to run in the middleware chain (a good place to put it might be right before `cookieParser`):

E.g. in `config/http.js` :

```
// .....
module.exports.http = {

  middleware: {

    p3p: require('p3p')(p3p.recommmended), // <==== set up the custom middleware here and

    order: [
      'startRequestTimer',
      'p3p', // <============ configured the order of our "p3p" custom middleware here
      'cookieParser',
      'session',
      'bodyParser',
      'handleBodyParserError',
      'compress',
      'methodOverride',
      'poweredBy',
      '$custom',
      'router',
      'www',
      'favicon',
      '404',
      '500'
    ],
    // .....
  }
};
```

Check out the examples below for more guidance - and be sure and follow the links to see the docs for the module you're using for the latest information, comparative analysis of its features, any recent bug fixes, and advanced usage details.

Using node-p3p

> `node-p3p` is open-source under the MIT license.

```
# In your sails app
npm install p3p --save
```

Then in the `middleware` config object in `config/http.js` :

```
  // ...
  // node-p3p provides a recommended compact privacy policy out of the box
  p3p: require('p3p')(require('p3p').recommended)
  // ...
```

Using lusca

> `lusca` is open-source under the Apache license

```
# In your sails app
npm install lusca --save
```

Then in the `middleware` config object in `config/http.js` :

```
  // ...
  // "ABCDEF" ==> The compact privacy policy to use.
  p3p: require('lusca').p3p('ABCDEF')
  // ...
```

Additional Resources:

- Description of the P3P Project (Microsoft)
- "P3P Work suspended" (W3C)
- P3P Compact Specification (CompactPrivacyPolicy.org)

Socket Hijacking

Unfortunately, cross-site request forgery attacks are not limited to the HTTP protocol. WebSocket hijacking (sometimes known as CSWSH) is a commonly overlooked vulnerability in most realtime applications. Fortunately, since Sails treats both HTTP and WebSocket requests as first-class citizens, its built-in CSRF protection and configurable CORS rulesets apply to both protocols.

You can prepare your Sails app against CSWSH attacks by enabling the built-in protection in `config/csrf.js` and ensuring that a `_csrf` token is sent with all relevant incoming socket requests. Additionally, if you're planning on allowing sockets to connect to your Sails app cross-origin (i.e. from a different domain, subdomain, or port) you'll want to configure your CORS settings accordingly. You can also define the `authorization` setting in `config/sockets.js` as a custom function which allows or denies the initial socket connection based on your needs.

Notes

- CSWSH prevention is only a concern in scenarios where people use the same client application to connect sockets to multiple web services (e.g. cookies in a browser like Google Chrome can be used to connect a socket to Chase.com from both Chase.com and Horrible-Hacker-Site.com.)

HTTP Strict Transport Security

Strict Transport Security (STS) is an opt-in security enhancement that forces usage of `HTTPS` instead of `HTTP`.

Enabling STS

Implementing STS is actually very simple and only takes a few lines of code. But better yet, a few different open-source modules exist that bring support for this feature to Express and Sails. To use one of these modules, install it from npm using the directions below, then open `config/http.js` in your project and configure it as a custom middleware. The example(s) below cover basic usage and configuration. For more guidance and advanced usage details, be sure and follow the link to the docs.

Using lusca

> `lusca` is open-source under the Apache license

```
# In your sails app
npm install lusca --save
```

Then in the `middleware` config object in `config/http.js`:

```
  // ...
  // maxAge ==> Number of seconds strict transport security will stay in effect.
  strictTransportSecurity: require('lusca').hsts({ maxAge: 31536000 })
  // ...
```

Additional Resources

- HTTP Strict Transport Security (OWasp)

XSS

Cross-site scripting (XSS) is a type of attack in which a malicious agent manages to inject client-side JavaScript into your website, so that it runs in the trusted environment of your users' browsers.

Additional Resources

- [XSS (OWasp)][https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)]
- [XSS Prevention Cheatsheet][https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet]

Services

Services can be thought of as libraries which contain functions that you might want to use in many places of your application. For example, you might have an EmailService which wraps some default email message boilerplate code that you would want to use in many parts of your application. The main benefit of using services in Sails is that they are *globalized*--you don't have to use `require()` to access them.

Creating a Service?

Simply save a Javascript file containing a function or object into your **api/services** folder. The filename will be used as the globally-accessible variable name for the service. For example, an email service might look something like this:

```
// EmailService.js - in api/services
module.exports = {

    sendInviteEmail: function(options) {

        var opts = {"type":"messages","call":"send","message":
            {
                "subject": "YourIn!",
                "from_email": "info@balderdash.co",
                "from_name": "AmazingStartupApp",
                "to":[
                    {"email": options.email, "name": options.name}
                ],
                "text": "Dear "+options.name+",\nYou're in the Beta! Click <insert link>
            }
        };

        myEmailSendingLibrary.send(opts);

    }
};
```

You can then use `EmailService` anywhere in your app:

```
// Somewhere in a controller
  EmailService.sendInviteEmail({email: 'test@test.com', name: 'test'});
```

Understanding Sessions in Sails

For our purposes **sessions** are synonymous with a few components that together allow you to store information about a user agent between requests.

> A **user agent** is the software (e.g. browser or native application) that represents you on a device (e.g. a browser tab on your computer, a smartphone application, or your refrigerator). It is associated one-to-one with a cookie or access token.

Sessions can be very useful because the request/response cycle is **stateless**. The request/response cycle is considered stateless because neither the client nor the server inherently stores any information between different requests about a particular request. Therefore the lifecycle of a request/response ends when a response is made to the requesting user agent (e.g. `res.send()`).

Note, we’re going to discuss sessions in the context of a browser user agent. While you can use sessions in Sails for whatever you like, it is generally a best practice to use it purely for storing the state of user agent authentication. Authentication is a process that allows a user agent to prove that they have a certain identity. For example, in order to access some protected functionality, I might need to prove that my browser tab actually corresponds with a particular user record in a database. If I provide you with a unique name and a password, you can look up the name and compare my password with a stored (hopefully encyrpted) password. If there's a match, I'm authenticated. But how do you store that "authenticated-ness" between requests? That's where sessions come in.

What sessions are made of

There are three main components to the implementation of sessions in Sails:

1. the **session store** where information is retained
2. the middleware that manages the session
3. a cookie that is sent along with every request and stores a session id (by default, `sails.sid`)

The **session store** can either be in memory (e.g. the default Sails session store) or in a database (e.g. Sails has built-in support for using Redis for this purpose). Sails builds on top of Connect middleware to manage the session; which includes using a **cookie** to store a session id (`sid`) on theuser agent.

A day in the life of a *request*, a *response*, and a *session*

When a `request` is sent to Sails, the request header is parsed by the session middleware.

Scenario 1: The request header has no *cookie property*

If the header does not contain a cookie property, a `sid` is created in the session and a default session dictionary is added to `req` (e.g. `req.session`). At this point you can make changes to the session property (usually in a controller/action). For example, let's look at the following *login* action.

```
module.exports = {

  login: function(req, res) {

    // Authentication code here

    // If successfully authenticated

    req.session.userId = foundUser.id;   // returned from a database

    return res.json(foundUser);

  }
}
```

Here we added a `userId` property to `req.session`.

> **Note:** The property will not be stored in the *session store* nor available to other requests until the response is sent.

Once the response is sent, any new requests will have access to `req.session.userId`. Since we didn't have a cookie *property* in the request header a cookie will be established for us.

Scenario 2: The request header has a cookie *property* with a `Sails.sid`

Now when the user agent makes the next request, the `Sails.sid` stored on the cookie is checked for authenticity and if it matches an existing `sid` in the session store, the contents of the session store is added as a property on the `req` dictionary (e.g. `req.session`). We can access properties on `req.session` (e.g. `req.session.me`) or add properties to it (e.g. `req.session.me == someValue`). The values in the session store might change but generally the `Sails.sid` and `sid` do not change.

When does the `Sails.sid` change?

By default, the Sails session store is *in memory*. Therefore, when you close the Sails server, the current session store moves on to session heaven (e.g. the session store disappears). When Sails is restarted, although a user agent request contains a `Sails.sid` in the cookie,

the `sid` is no longer in the session store. Therefore, a new `sid` will be generated and replaced in the cookie. The `Sails.sid` will also change if the user agent cookie expires or is removed.

> The lifespan of a Sails cookie can be changed from its default setting (e.g. never expires) to a new setting by accessing the `cookie.maxAge` property in `projectName/config/session.js`.

Using *Redis* as the session store

Redis is a key-value database package that can be used as a session store that is separate from the Sails instance. This configuration for sessions has two benefits. The first is that the session store will remain viable between Sails restarts. The second is that if you have multiple Sails instances behind a load balancer, all of the instances can point to a single consolidated session store.

To enable Redis as a session store open `projectName/config/session.js` in your favorite text editor and uncomment the `adapter` property. That's it. During development as long as you have a Redis instance running on the same machine as your Sails instance your session store will use Redis. You can point to a different Redis instance by configuring the following optional properties in `projectName/config/session.js`:

```
// host: 'localhost',
// port: 6379,
// ttl: <redis session TTL in seconds>,
// db: 0,
// pass: <redis auth password>,
// prefix: 'sess:',
```

For more information on configuring these properties go to https://github.com/tj/connect-redis.

Nerdy details of how the session cookie is created

The value for the cookie is created by first hashing the `sid` with a configurable *secret* which is just a long string.

> You can change the session `secret` property in `projectName/config/session.js`.

The Sails `sid` (e.g. `Sails.sid`) then becomes a combination of the plain `sid` followed by a hash of the `sid` plus the `secret`. To take this out of the world of abstraction, let's use an example. Sails creates a `sid` of `234lj232hg234jluy32UUYUHH` and a `session secret` of `9238cca11a83d473e10981c49c4f`. These values are simply two strings that Sails combine and

hash to create a `signature` of `AuSosBAbL9t3Ev44EofZtIpiMuV7fB2oi`. So the `Sails.sid` becomes `234lj232hg234jluy32UUYUHH.AuSosBAbL9t3Ev44EofZtIpiMuV7fB2oi` and is stored in the user agent cookie by sending a `set-cookie` property in the response header.

What does this prevent? It prevents a user from guessing the `sid` as well as prevents a evil doer from spoofing a user into making an authetication request with a `sid` that the evil doer knows. This could allow the evil doer to use the `sid` to do bad things while the user is authenticated via the session.

Testing your code

Preparation

For our test suite, we use mocha. Before you start building your test cases, you should first organise your `./test` directory structure, for example in the following way:

```
./myApp
├── api
├── assets
├── ...
├── test
│   ├── unit
│   │   ├── controllers
│   │   │   └── UsersController.test.js
│   │   ├── models
│   │   │   └── Users.test.js
│   │   └── ...
│   ├── fixtures
│   ├── ...
│   ├── bootstrap.test.js
│   └── mocha.opts
└── views
```

bootstrap.test.js

This file is useful when you want to execute some code before and after running your tests(e.g. lifting and lowering your sails application). Since your models are converted to waterline collections on lift, it is necessary to lift your sailsApp before trying to test them (This applies similarly to controllers and other parts of your app, so be sure to call this file first).

```
var Sails = require('sails'),
  sails;

before(function(done) {

  // Increase the Mocha timeout so that Sails has enough time to lift.
  this.timeout(5000);

  Sails.lift({
    // configuration for testing purposes
  }, function(err, server) {
    sails = server;
    if (err) return done(err);
    // here you can load fixtures, etc.
    done(err, sails);
  });
});

after(function(done) {
  // here you can clear fixtures, etc.
  Sails.lower(done);
});
```

mocha.opts

This file should contain mocha configuration as described here: mocha.opts

Note: If you are writing your test in CoffeeScript be sure to add these lines to your `mocha.opts`.

```
--require coffee-script/register
--compilers coffee:coffee-script/register
```

Note: The default test-case timeout in Mocha is 2 seconds. Increase the timeout value in mocha.opts to make sure the sails lifting completes before any of the test-cases can be started. For example:

```
--timeout 5s
```

Writing tests

Once you have prepared your directory you can start writing your unit tests.

./test/unit/models/Users.test.js

```
describe('UsersModel', function() {

  describe('#find()', function() {
    it('should check find function', function (done) {
      Users.find()
        .then(function(results) {
          // some tests
          done();
        })
        .catch(done);
    });
  });

});
```

Testing controllers

To test controller responses you can use Supertest library which provides several useful methods for testing HTTP requests.

./test/unit/controllers/UsersController.test.js

```
var request = require('supertest');

describe('UsersController', function() {

  describe('#login()', function() {
    it('should redirect to /mypage', function (done) {
      request(sails.hooks.http.app)
        .post('/users/login')
        .send({ name: 'test', password: 'test' })
        .expect(302)
        .expect('location','/mypage', done);
    });
  });

});
```

Running tests

In order to run your test using mocha, you'll have to use `mocha` in the command line and then pass as arguments any test you want to run, be sure to call bootstrap.test.js before the rest of your tests like this `mocha test/bootstrap.test.js test/unit/**/*.test.js`

Using `npm test` to run your test

To avoid typing the mocha command, like stated before (specially when calling bootstrap.test.js) and using `npm test` instead, you'll need to modify your package.json. On the scripts obj, add a `test` key and type this as its value `mocha test/bootstrap.test.js test/unit/**/*.test.js` like this:

```
// package.json
scripts": {
   "start": "node app.js",
   "debug": "node debug app.js",
   "test": "mocha test/bootstrap.test.js test/unit/**/*.test.js"
 },
// More config
```

The `*` is a wildcard used to match any file inside the `unit` folder that ends in `.test.js` so if it suits you, you can perfectly modify it to search for `*.spec.js` instead. In the same way you can use wildcards for your folders by using two `*` instead of one.

Code coverage

Another popular method for testing your code is Code Coverage.

You can use mocha and istanbul to check your code and prepare various coverage reports (HTML, Cobertura) which can be used in continuous integration services such as Jenkins.

To test your code and prepare a simple HTML report run the following commands:

```
istanbul cover -x "**/config/**" _mocha -- --timeout 5000
istanbul report html
```

Upgrading

See the migration guide for details on upgrading to Sails v0.11.x!

Upgrading to v0.10

For the most part, running sails lift in an existing v0.9 project should just work. The core contributors have taken a number of steps to make the upgrade as easy as possible, and if you follow the deprecation messages in the console, you should do just fine.

Sails v0.10 comes with some big changes. The sections below provide a high level overview of what's changed, major bug fixes, enhancements and new features, as well as a basic tutorial on how to upgrade your v0.9.x Sails app to v0.10.

File uploads

The Connect multipart middleware will soon be officially deprecated. But since this module was used as the built-in HTTP body parser in Sails v0.9 and Express v3, this is a breaking change for v0.9 Sails projects relying on `req.files`.

By default in v0.10, Sails includes skipper, a body parser which allows for streaming file uploads without buffering tmp files to disk. For run-of-the-mill file upload use cases, Skipper comes with bundled support for uploads to local disk (via skipper-disk), but streaming uploads can be plugged in to any of its supported adapters.

For examples/documentation, please see the Skipper repository as well as the Sails documentation on `req.file()`.

Why?

A body parser's job is to parse the "body" of incoming multipart HTTP requests. Sometimes, that "body" includes text parameters, but sometimes, it includes file uploads.

Connect multipart is great code, and it supports both file uploads AND text parameters in multipart requests. But like most modules of its kind, it accomplishes this by buffering file uploads to disk. This can quickly overwhelm a server's available disk space, and in many cases exposes a serious DoS attack vulnerability.

Skipper is unique in that it supports **streaming** file uploads, but also maintains support for metadata in the request body (i.e. JSON/XML/urlencoded request body parameters). It uses a handful of heuristics to make sure only the files you're expecting get plugged in and received by the blob adapter, and other (potentially malicous) file fields are ignored.

> ### Important!
>
> For Skipper to work, you *must include all text parameters BEFORE file parameters* in file upload requests to the server. Once Skipper sees the first file field, it stops waiting for text parameters (this is to avoid unnecessary/unsafe buffering of file data).

Configuring a different body parser

As with most things in Sails, you can use any Connect/Express/Sails-compatible bodyparser you like. To switch back to **connect-multipart**, or any other body parser (like **formidable** or **busboy**), change your app's http configuration.

Blueprints

A new blueprint action (`findOne`) has been added. For instance, if you have a `FooController` and `Foo` model, then send a request to `/foo/5` , the `findOne` action in your `FooController` will run. If you don't have a `findOne` action, the `findOne` blueprint action will be used in its stead. Requests sent to `/foo` will still run the find controller/blueprint action.

Policies

Policies work exactly as they did in v0.9- however there is a new consideration you should take into account: Due to the introduction of the more specific `findOne()` blueprint action mentioned above, you will want to make sure you're handling it explicitly in your policy mapping configuration.

For example, let's say you have a v0.9 app whose `policies.js` configuration prevents access to the `find` action in your `DoveController` :

```
module.exports.policies = {
  '*': true,
  DoveController: {
    find: false
  }
};
```

Assuming rest blueprint routes are enabled, this would prevent access to requests like both `/dove` and `/dove/14` . But now in v0.10, since `/dove/14` will actually run the `findOne` action, we must handle it explicitly:

```
module.exports.policies = {
  '*': true,
  DoveController: {
    find: false,
    findOne: false
  }
};
```

Pubsub

Summary

- `message` socket (i.e. "comment") event on client is now `modelIdentity` (where "modelIdentity" is different depending on the model that the `publish*()` method was called from.
- Clients are no longer subscribed to model-creation events by the blueprint routes. To listen for creation events, use `Model.watch()` .
- The events that were formerly `create` , `update` , and `destroy` are now `created` , `updated` , and `destroyed` .

Details

The biggest change to pubsub is that Socket.io events are emitted under the name of the model emitting them. Previously, your client listened for the `message` event and then had to determine which model it came from based on the included data:

```
socket.on('message', function(cometEvent) {
   if (cometEvent.model == 'user') {
     // Handle inbound messages related to a user record
   }
   else if (cometEvent.model === 'product') {
     // Handle inbound messages related to a product record
   }
   // ...
}
```

Now, you subscribe to the identity of the model:

```
socket.on('user', function(cometEvent) {
  // Handle inbound messages related to a user record
});

socket.on('product', function (cometEvent) {
  // Handle inbound messages related to a product record
});
```

This helps to structure your front end code.

The way you subscribe clients to models has also changed. Previously, you specified whether you were subscribing to the model class (class room) or one or more model instances based on the parameters that you passed to `Model.subscribe` . It was effectively one method to do two very different things.

Now, you use `Model.subscribe()` to subscribe only to model instances (records). You can also specify event "contexts", or types, that you'd like to hear about. For example, if you only wanted to get messages about updates to an instance, you would call `User.subscribe(req, myUser, 'update')` . If no context is given in a call to `.subscribe()` , then all contexts specified by the model class's autosubscribe property will be used.

To subscribe to model creation events, you can now use `Model.watch()` . Upon subscription, your clients will receive messages every time a new record is created on that model using the blueprint routes, and will automatically be subscribed to the new instance as well.

Remember, when working with blueprints, clients are no longer auto subscribed to the class room. This must be done manually.

Finally, if you want to see all pubsub messages from all models, you can access the `firehose` , a development-only tool that broadcasts messages about *everything* that happens to your models. You can subscribe to the firehose using `sails.sockets.subscribeToFirehose(socket)` , or on the front end by making a socket request to `/firehose` . The firehose will broadcast a `firehose` event whenever a model is created, updated, destroyed, added to, removed from or messaged. This effectively replaces the `message` event used in previous Sails versions.

To see examples of the new pubsub methods in action, see SailsChat.

Arguments to lifecycle callbacks are now typecasted

Previously, with `schema: true`, if you sent an attribute value to a `.create()` or `.update()` that did not match the expected type declared in the model's attributes, the value you passed in would still be accessible in your model's lifecycle callbacks.

In Sails/Waterline v0.10, this is no longer the case. Values passed to `.create()` and `.update()` are type-casted before your lifecycle callbacks run. Affected lifecycle callbacks include `beforeUpdate()`, `beforeCreate()`, and `beforeValidate()`.

beforeValidation() is now beforeValidate()

If you were using the `beforeValidation` or `afterValidation` model lifecycle callbacks in any of your models, you should change them to `beforeValidate` or `afterValidate`. This change was made in Waterline to match the style of the other lifecycle callbacks (e.g. `beforeCreate`, `afterUpdate`, etc.).

.done() vs. .exec()

The old (/confusing?) meaning of `.done()` has been deprecated.

In Sails <= v0.8, the syntax for executing an ORM query was `Model. [ … ] .done( cb )`. In v0.9, when promise support was added, the `Model. [ … ] .exec( cb )` became the recommended replacement, since `.done()` has a special meaning in the promise spec. However, the original usage of `.done()` was left untouched to make upgrading from v0.8 to v0.9 easier.

But as of Sails/Waterline v0.10, the original meaning of `.done()` has been officially deprecated to allow for a more robust promise implementation going forward, and pluggable promise library support (e.g. choose `Q` or `Bluebird` etc.).

Associations

Sails v0.10 introduces associations between data models. Since the work we've done on associations is largely additive, your existing models should still just work. That said, this is a powerful new feature that allows you to write less code and makes your app more maintainable, so we suggest taking advantage of it! To learn about how to use associations in Sails, check out the docs.

Associations (or "relations") are really just special attributes. Instead of string or integer values, you can specify an instance of a model or a collection of model instances. You can think about this kind of like an object (`{...}`) or an array (`[{...}, {...}]`) you might store

as JSON in a NoSQL database. The difference is, in Sails, this works with any of the supported databases, and even allows you to populate (i.e. join) across different databases and types of databases.

Generators

Sails has had support for generating code for a while now (e.g. `sails generate controller foo`) but in v0.10, we wanted to make this feature more extensible, open, and accessible to everybody in the Sails community. With that in mind, v0.10 comes with a complete rewrite of the command-line tool, and pluggable generators. Want to be able to run `sails generate blog foo` to make a new blog built on Sails? Create a `blog` generator (run sails `generate generator blog`), add your templates, and configure the generator to copy the new templates over. Then you can release it to the community by publishing an npm module called `sails-generate-blog` . Compatibility with Yeoman generators is also in our roadmap.

Command-line tool

The big change here is how you create a new api. In the past you called `sails generate new_api` . This would generate a new controller and model called `new_api` in the appropriate places. This is now done using `sails generate api new_api` .

You can still generate models and controllers seperately using the same CLI Commands.

Also, `--linker` switch is no longer available. In previous version, if `--linker` switch was provided, it created a `myApp/assets/linker folder` , with `js` , `styles` and `templates` folders inside. In this new version, the `myApp/assets/linker` folder is not created. Compiling CoffeeScript and Less is the default behavior now, right from the `myApp/assets/js` and `myApp/assets/scripts` folders.

Custom server responses

In v0.10, you can now generate your own custom server responses.

Like before, there are a few that we automatically create for you. Instead of generating `myApp/config/500.js` and other `.js` responses in the config directory, they are now generated in `myApp/api/responses/` .

To migrate, you will need to create a new v0.10 project and copy the `myApp/api/responses` directory into your existing app. You will then modify the appropriate .js file to reflect any customization you made in your response logic files (500.js,etc).

Legacy data stored in the temporary sails-disk database

`sails-disk`, used by default in new Sails projects, now stores data a bit differently. If you have some temporary data stored in a 0.9.x project, you'll want to wipe it out and start fresh. To do this:

From your project's root directory:

```
$ rm .tmp/disk.db
```

Adapter/Database Configuration

`config.adapters` (in `myApp/config/adapters.js`) is now config.connections (in new projects, this is generated in `myApp/config/connections.js`). Also, `config.model` is now `config.models`.

Your app's default `connection` (i.e. database) should now be configured as a string `config.models.connection` used by default for model. New projects are generated with a `/config/models.js` file that includes the default connection.

To configure a model to use specific adapters, you must now specify them in the `connection` key instead of `adapters`.

For example:

```
module.exports = {

    connection: ['someMongoDatabase'],

    attributes: {
        name:{
            type     : 'string',
            required : true
        }
    }
};
```

Blueprints/Controller configuration

The object literal describing controller configuration overrides for controller blueprints should change from:

```
...
_config: {
  blueprints: {
    rest: true,
    ...
  }
}
```

to:

```
...
_config: {
    rest: true,
    ...
}
```

Layout paths:

In Sails v0.9, you could use the following syntax to specify `auth/someLayout.ejs` as a custom layout when rendering a view:

```
return res.view('auth/login',{
  layout: 'someLayout'
});
```

However in Sails v0.10, all layout paths are relative to your app's views path. In other words, the relative path of the layout is no longer resolved from the view's own path-- it is now always resolved from the views path. This makes it easier to understand which file is being used, particularly when layout files have similar names:

```
return res.view('auth/login', {
  layout: 'auth/someLayout'
});
```

To v0.11

v0.11 comes with many minor improvements, as well as some internal cleanup in core. The biggest change is that Sails core is now using Socket.io v1.

Almost none of this should affect the existing code in project, but there are a few important differences and new features to be aware of. See the full migration guide for details on how to upgrade.

Views

Overview

In Sails, views are markup templates that are compiled *on the server* into HTML pages. In most cases, views are used as the response to an incoming HTTP request, e.g. to serve your home page.

Alternatively, a view can be compiled directly into an HTML string for use in your backend code (see `sails.renderView()`.) For instance, you might use this approach to send HTML emails, or to build big XML strings for use with a legacy API.

Creating a view

By default, Sails is configured to use EJS (Embedded Javascript) as its view engine. The syntax for EJS is highly conventional- if you've worked with php, asp, erb, gsp, jsp, etc., you'll immediately know what you're doing.

If you prefer to use a different view engine, there are a multitude of options. Sails supports all of the view engines compatible with Express via Consolidate.

Views are defined in your app's `views/` folder by default, but like all of the default paths in Sails, they are configurable. If you don't need to serve dynamic HTML pages at all (say, if you're building an API for a mobile app), you can remove the directory from your app.

Compiling a view

Anywhere you can access the `res` object (i.e. a controller action, custom response, or policy), you can use `res.view` to compile one of your views, then send the resulting HTML down to the user.

You can also hook up a view directly to a route in your `routes.js` file. Just indicate the relative path to the view from your app's `views/` directory. For example:

```
{
  'get /': {
    view: 'homepage'
  },
  'get /signup': {
    view: 'signupFlow/basicInfo'
  },
  'get /signup/password': {
    view: 'signupFlow/chooseAPassword'
  },
  // and so on.
}
```

What about single-page apps?

If you are building a web application for the browser, part (or all) of your navigation may take place on the client; i.e. instead of the browser fetching a new HTML page each time the user navigates around, the client-side code preloads some markup templates which are then rendered in the user's browser without needing to hit the server again directly.

In this case, you have a couple of options for bootstrapping the single-page app:

- Use a single view, e.g. `views/publicSite.ejs` . Advantages:
  - You can use the view engine in Sails to pass data from the server directly into the HTML that will be rendered on the client. This is an easy way to get stuff like user data to your client-side javascript, without having to send AJAX/WebSocket requests from the client.
- Use a single HTML page in your assets folder , e.g. `assets/index.html` . Advantages:
  - Although you can't pass server-side data directly to the client this way, this approach allows you to further decouple the client and server-side parts of your application.
  - Anything in your assets folder can be moved to a static CDN (like Cloudfront or CloudFlare), allowing you to take advantage of that provider's geographically distributed data centers to get your content closer to your users.

Layouts

When building an app with many different pages, it can be helpful to extrapolate markup shared by several HTML files into a layout. This reduces the total amount of code in your project and helps you avoid making the same changes in multiple files down the road.

In Sails and Express, layouts are implemented by the view engines themselves. For instance, `jade` has its own layout system, with its own syntax.

For convenience, Sails bundles special support for layouts **when using the default view engine, EJS**. If you'd like to use layouts with a different view engine, check out that view engine's documentation to find the appropriate syntax.

Creating Layouts

Sails layouts are special `.ejs` files in your app's `views/` folder you can use to "wrap" or "sandwich" other views. Layouts usually contain the preamble (e.g. `!DOCTYPE html<html><head>....</head><body>`) and conclusion (`</body></html`). Then the original view file is included using `<%- body %>`. Layouts are never used without a view- that would be like serving someone a bread sandwich.

Layout support for your app can be configured or disabled in `config/views.js`, and can be overridden for a particular route or action by setting a special local called `layout`. By default, Sails will compile all views using the layout located at `views/layout.ejs`.

To specify what layout a view uses, see the example below. There is more information in the docs at routes.

The example route below will use the view located at `./views/users/privacy.ejs` within the layout located at `./views/users.ejs`

```
'get /privacy': {
    view: 'users/privacy',
    locals: {
      layout: 'users'
    }
  },
```

The example controller action below will use the view located at `./views/users/privacy.ejs` within the layout located at `./views/users.ejs`

```
privacy: function (req, res) {
  res.view('users/privacy', {layout: 'users'})
}
```

Notes

> ### Why do layouts only work for EJS?
>
> In Express 3, built-in support for layouts/partials was deprecated. Instead, developers are expected to rely on the view engines themselves to implement this features. (See https://github.com/balderdashy/sails/issues/494 for more info on that.)
>
> Since adopting Express 3, Sails has chosen to support the legacy `layouts` feature for convenience, backwards compatibility with Express 2.x and Sails 0.8.x apps, and in particular, familiarity for new community members coming from other MVC frameworks. As a result, layouts have only been tested with the default view engine (ejs).
>
> If layouts aren't your thing, or (for now) if you're using a server-side view engine other than ejs, (e.g. Jade, handlebars, haml, dust) you'll want to set `layout:false` in `sails.config.views`, then rely on your view engine's custom layout/partial support.

Locals

The variables accessible in a particular view are called `locals`. Locals represent server-side data that is *accessible* to your view-- locals are not actually *included* in the compiled HTML unless you explicitly reference them using special syntax provided by your view engine.

```
Logged in as <%= name="" %="">.
```

Using locals in your views

The notation for accessing locals varies between view engines. In EJS, you use special template markup (e.g. `<%= someValue %>`) to include locals in your views.

There are three kinds of template tags in EJS:

- `<%= someValue %>`
  - HTML-escapes the `someValue` local, and then includes it as a string.
- `<%- someRawHTML %>`
  - Includes the `someRawHTML` local verbatim, without escaping it.
  - Be careful! This tag can make you vulnerable to XSS attacks if you don't know what you're doing.
- `<% if (!loggedIn) { %> <a>Logout</a> <% } %>`
  - Runs the javascript inside the `<% ... %>` when the view is compiled.
  - Useful for conditionals (`if` / `else`), and looping over data (`for` / `each`).

Here's an example of a view (`views/backOffice/profile.ejs`) using two locals, `user` and `corndogs` :

```
<div>
  <h1><%= user.name %>'s first view</h1>
  <h2>My corndog collection:</h2>
  <ul>
    <% _.each(corndogs, function (corndog) { %>
    <li><%= corndog.name %></li>
    <% }) %>
  </ul>
</div>
```

> You might have noticed another local, `_`. By default, Sails passes down a few locals to your views automatically, including lodash (`_`).

If the data you wanted to pass down to this view was completely static, you don't necessarily need a controller- you could just hard-code the view and its locals in your `config/routes.js` file, i.e:

```
// ...
'get /profile': {
  view: 'backOffice/profile',
  locals: {
    user: {
      name: 'Frank',
      emailAddress: 'frank@enfurter.com'
    },
    corndogs: [
      { name: 'beef corndog' },
      { name: 'chicken corndog' },
      { name: 'soy corndog' }
    ]
  }
},
// ...
```

On the other hand, in the more likely scenario that this data is dynamic, we'd need to use a controller action to load it from our models, then pass it to the view using the res.view() method.

Assuming we hooked up our route to one of our controller's actions (and our models were set up), we might send down our view like this:

```
// in api/controllers/UserController.js...

  profile: function (req, res) {
    // ...
    return res.view('backOffice/profile', {
      user: theUser,
      corndogs: theUser.corndogCollection
    });
  },
  // ...
```

Partials

Sails uses `ejs-locals` in its view rendering code, so in your views you can do:

```
<%- partial ('foo.ejs') %>
```

to render a partial located at `/views/foo.ejs`. All of your locals will be sent to the partial automatically.

The paths are relative to the view that loads the partial. So if you have a a user view at `/views/users/view.ejs` and want to load `/views/partials/widget.ejs` then you would use:

```
<%- partial ('../partials/widget.ejs') %>
```

One thing to note: partials are rendered synchronously, so they will block Sails from serving more requests until they're done loading. It's something to keep in mind while developing your app, especially if you anticipate a large number of connections.

> NOTE: When using other templating languages than ejs, their syntax for loading partials or block, etc. will be used. Please refer to their documentation for more information on their syntax and conventions

View Engines

The default view engine in Sails is EJS.

Swapping out the view engine

To use a different view engine, you should use npm to install it in your project, then set `sails.config.views.engine` in `config/views.js`.

For example, to switch to jade, run `npm install jade --save-dev`, then set `engine: 'jade'` in `config/views.js`.

Supported view engines

- atpl
- dust (website) (.dust)
- eco
- ect (website)
- ejs (.ejs)
- haml (website)
- haml-coffee (website)
- handlebars (website) (.hbs)
- hogan (website)
- jade (website) (.jade)
- jazz
- jqtpl (website)
- JUST
- liquor
- lodash (website)
- mustache
- QEJS
- ractive
- swig (website)
- templayed
- toffee
- underscore (website)
- walrus (website)
- whiskers

Adding new custom view engines

For instructions on adding support for a view engine not listed above, check out the consolidate project repository.

Extending Sails

In keeping with the Node philosophy, Sails aims to keep its core as small as possible, delegating all but the most critical functions to separate modules*. There are currently three types of extensions that you can make to Sails:

- **Generators** - for adding and overriding functionality in the Sails CLI. *Example*: sails-generate-model, which allows you to create models on the command line with `sails generate model foo`.
- **Adapters** - for integrating Waterline (Sails' ORM) with new data sources, including databases, APIs, or even hardware. *Example*: sails-postgresql, the official PostgreSQL adapter for Sails.
- **Hooks** - for overriding or injecting new functionality in the Sails runtime. *Example*: sails-hook-autoreload, which adds auto-refreshing for a Sails project's API without having to manually restart the server.

If you're interested in developing a “plugin” for Sails, you will most often want to make a hook.

* Some of the more important modules, like the Request hook, are still bundled with the Sails core, but they could technically be removed and installed separately.

Adapters

Status

Stability: 3 - Stable

The API has proven satisfactory, but cleanup in the underlying code may cause minor changes. Backwards-compatibility is guaranteed.

What is an adapter?

Adapters expose **interfaces**, which imply a contract to implement certain functionality. This allows us to guarantee conventional usage patterns across multiple models, developers, apps, and even companies, making app code more maintainable, efficient, and reliable. Adapters are useful for integrating with databases, open APIs, internal/proprietary web services, or even hardware.

What kind of things can I do in an adapter?

Adapters are mainly focused on providing model-contextualized CRUD methods. CRUD stands for create, read, update, and delete. In Sails/Waterline, we call these methods `create()`, `find()`, `update()`, and `destroy()`.

For example, a `MySQLAdapter` implements a `create()` method which, internally, calls out to a MySQL database using the specified table name and connection informtion and runs an `INSERT ...` SQL query.

In practice, your adapter can really do anything it likes-- any method you write will be exposed on the raw connection objects and any models which use them.

Class methods

Below, `class methods` refer to the static, or collection-oriented, functions available on the model itself, e.g. `User.create()` or `Menu.update()`. To add custom class methods to your model (beyond what is provided in the adapters it implements), define them as top-level key/function pairs in the model object.

Instance methods

`instance methods` on the other hand, (also known as object, or model, methods) refer to methods available on the individual result models themselves, e.g. `User.findOne(7).done(function (err, user) { user.someInstanceMethod(); });`. To add custom instance methods to your model (beyond what is provided in the adapters it implements), define them as key/function pairs in the `attributes` object of the model's definition.

DDL and auto-migrations

`DDL` stands for data-definition language, and is a common fixture of schema-oriented databases. In Sails, auto-migrations are supported out of the box. Since adapters for the most common SQL databases support `alter()`, they also support automatic schema migration! In your own adapter, if you write the `alter()` method, the same behavior will take effect. The feature is configurable using the `migrate` property, which can be set to `safe` (don't touch the schema, period), `drop` (recreate the tables every time the app starts), or `alter` (the default-- merge the schema in the apps' models with what is currently in the database).

List of Available Adapters

This file is meant to be an up to date, comprehensive list of all of the adapters available for the Sails.js framework. If we missed one or you would like to add an adapter you made, just submit a Pull Request to this file, adding to the list.

Officially Supported Adapters

sails-disk

https://github.com/balderdashy/sails-disk/

Write to your computer's hard disk, or a mounted network drive. Not suitable for at-scale production deployments, but great for a small project, and essential for developing in environments where you may not always have a database set up. This adapter is bundled with Sails and works out of the box with zero configuration.

Interfaces implemented:

- Semantic
- Queryable
- Streaming

sails-memory

https://github.com/balderdashy/sails-memory/

Pretty much like Disk, but doesn't actually write to disk, so it's not persistent. Not suitable for at-scale production deployments, but useful when developing on systems with little or no disk space.

Interfaces implemented:

- Semantic
- Queryable
- Streaming

sails-mysql

https://github.com/balderdashy/sails-mysql/

MySQL is the world's most popular relational database. http://en.wikipedia.org/wiki/MySQL

Interfaces implemented:

- Semantic
- Queryable
- Streaming
- Migratable

sails-postgres

https://github.com/balderdashy/sails-postgresql/

PostgreSQL is another popular relational database.

Interfaces implemented:

- Semantic
- Queryable
- Streaming
- Migratable

sails-mongo

https://github.com/balderdashy/sails-mongo/

MongoDB is the leading NoSQL database.

Interfaces implemented:

- Semantic
- Queryable
- Streaming

sails-redis

https://github.com/balderdashy/sails-redis/

Redis is an open source, BSD licensed, advanced key-value store.

Interfaces implemented:

- Semantic
- Queryable

Community Supported Adapters

sails-orientdb

https://github.com/appscot/sails-orientdb

OrientDB is an Open Source NoSQL DBMS with the features of both Document and Graph DBMSs.

Interfaces implemented:

- Semantic
- Queryable
- Associations
- Migratable

sails-filemaker

https://github.com/geistinteractive/sails-filemaker

FileMaker, is cross platform relational database and development platform. It has been owned and published by Apple since 1988.

Interfaces implemented:

- Semantic

Have you written a Sails adapter? Submit a PR to this file and add it here!

Custom Adapters

Overview

Sails makes it fairly easy to write your own adapter. Check out the sails-boilerplate-repo to learn how

Generators

Status

Stability: 2 - Unstable

The API is in the process of settling, but has not yet had sufficient real-world testing to be considered stable.

Backwards-compatibility will be maintained if reasonable.

Purpose

What is my purpose in this world?

old partial content from when spec was an itty bitty baby

Generators are designed to make it easier to customize the `sails new` and `sails generate` command-line tools, and provide better support for different Gruntfiles, configuration options, view engines, coffeescript, etc.

Structure

A generator has either:

(1) a `generate` method, or

(2) a `configure` + `render` method (render may be omitted in the simplest of cases)

Sails

```
app (appPath + name)
    <- view
    <- folder
    <- jsonfile
    <- file

api (appPath + name)
    <- controller
    <- model

controller (appPath + template + name)
    <- file

model (appPath + template + name)
    <- file

view (appPath + template + name)
    <- file

file (destination + name + template + data)

jsonfile (destination + name + data)

folder (destination + name)
```

Custom Generators

Overview

Here is some info on writing custom generators.

List of Available Adapters

This file is meant to be an up to date, comprehensive list of all of the generators available for the Sails.js framework. If we missed one or you would like to add a generator you've made, just submit a Pull Request to this file, adding to the list.

Officially Supported Generators

sails-generate-generator

Github Repo

https://github.com/balderdashy/sails-generate-generator/

On NPM

'npm install sails-generate-generator`

Description

Generate the boilerplates to make your own generator.

Community Supported Generators

Hooks

Status

> **Stability: 3 - Stable**

What is a hook?

A hook is a Node module that adds functionality to the Sails core. The hook specification defines the requirements a module must meet for Sails to be able to import its code and make the new functionality available. Because they can be saved separately from the core, hooks allow Sails code to be shared between apps and developers without having to modify the framework.

Hooks vs. Services

Hooks share some common features with Sails services. They both allow developers to store commonly used code in one location, and they both make new methods available globally to a Sails app. However, there are some key differences between the two concepts:

- Services cannot be saved independently of an app. While some types of hooks may be tied to a single app (see Project Hooks), other types can be developed independently of a Sails app and installed using `npm install`.
- Hooks have their own initialization system. This allows them to be more dynamic and configure themselves when Sails lifts.
- Hooks can add new routes to a Sails app before it lifts.

Services are still a good choice for code that is shared between multiple controllers or models in an app, but

- is unlikely to be reused in another app
- won't need to behave differently in different environments (e.g. development vs. production)

For all other reusable code, hooks are the way to go!

Types of hooks

There are three types of hooks available in Sails:

1. **Core hooks**. These hooks provide many of the common features essential to a Sails app, such as request handling, blueprint route creation, and database integration via Waterline. Core hooks are bundled with the Sails core and are thus available to every app. You will rarely have a need to call core hook methods in your code.
2. **Project hooks**. These are hooks that live in the `api/hooks` folder of a Sails app. Project hooks provide a way to take advantage of the features of the hook system for code that doesn't need to be shared between apps.
3. **Installable hooks**. These hooks are installed into an app's `node_modules` folder using `npm install`. Installable hooks allow developers in the Sails community to create and "plug-in"-like modules for use in Sails apps.

Read more

- Using hooks in your app
- The hook specification
- Creating a project hook
- Creating an installable hook

The Hook Specification

Overview

Each Sails hook is implemeted as a Javascript function that takes a single argument—a reference to the running `sails` instance—and returns an object with one or more of the keys described later in this document. So, the most basic hook would look like this:

```
module.exports = function myBasicHook(sails) {
   return {};
}
```

It wouldn't do much, but it would work!

Each hook should be saved in its own folder with the filename `index.js`. The folder name should uniquely identify the hook, and the folder can contain any number of additional files and subfolders. Extending the previous example, if you saved the file containing `myBasicHook` in a Sails project as `index.js` in the folder `api/hooks/my-basic-hook` and then lifted your app with `sails lift --verbose`, you would see the following in the output:

`verbose: my-basic-hook hook loaded successfully.`

Hook features

The following features are available to implement in your hook. All features are optional, and can be implemented by adding them to the object returned by your hook function.

- .defaults
- .configure()
- .initialize()
- .routes

Custom hook data and functions

Any other keys added to the object returned from the main hook function will be provided in the `sails.hooks[<hook name>]` object. This is how custom hook functionality is provided to end-users. Any data and functions that you wish to remain private to the hook can be added *outside* the returned object:

```
// File api/hooks/myhook/index.js
module.exports = function myHook(sails) {

   // This var will be private
   var foo = 'bar';

   // This var will be public
   this.abc = 123;

   return {

      // This function will be public
      sayHi: function (name) {
         console.log(greet(name));
      }

   };

   // This function will be private
   function greet (name) {
      return "Hi, " + name + "!";
   }

};
```

The public var and function above would be available as `sails.hooks.myhook.abc` and `sails.hooks.myhook.sayHi`, respectively.

`.configure()`

The `configure` feature provides a way to configure a hook after the `defaults` objects have been applied to all hooks. By the time a custom hook's `configure()` function runs, all user-level configuration and core hook settings will have been merged into `sails.config`. However, you should *not* depend on other custom hooks' configuration at this point, as the load order of custom hooks is not guaranteed.

`configure` should be implemented as a function with no arguments, and should not return any value. For example, the following `configure` function could be used for a hook that communicates with a remote API, to change the API endpoint based on whether the user set the hook's `ssl` property to `true`. Note that the hook's configuration key is available in `configure` as `this.configKey`:

```
configure: function() {

   // If SSL is on, use the HTTPS endpoint
   if (sails.config[this.configKey].ssl == true) {
      sails.config[this.configKey].url = "https://" + sails.config[this.configKey].domain
   }
   // Otherwise use HTTP
   else {
      sails.config[this.configKey].url = "http://" + sails.config[this.configKey].domain;
   }
}
```

The main benefit of `configure` is that all hook `configure` functions are guaranteed to run before any `initialize` functions run; therefore a hook's `initialize` function can examine the configuration settings of other hooks.

`.defaults`

The `defaults` feature can be implemented either as an object or a function which takes a single argument (see “using `defaults` as a function” below) and returns an object. The object you specify will be used to provide default configuration values for Sails. You should use this feature to specify default settings for your hook. For example, if you were creating a hook that communicates with a remote service, you may want to provide a default domain and timeout length:

```
{
   myapihook: {
      timeout: 5000,
      domain: "www.myapi.com"
   }
}
```

If a `myapihook.timeout` value is provided via a Sails configuration file, that value will be used; otherwise it will default to `5000`.

Namespacing your hook configuration

For project hooks, you should namespace your hook’s configuration under a key that uniquely identifies that hook (e.g. `myapihook` above). For installable hooks, you should use the special `__configKey__` key to allow end-users of your hook to change the configuration key if necessary. The default key for a hook using `__configKey__` is the hook name. For example, if you create a hook called `sails-hooks-myawesomehook` which includes the following `defaults` object:

```
{
   __configKey__: {
      name: "Super Bob"
   }
}
```

then it will, by default, provide default settings for the `sails.config.myawesomehook.name` value. If the end-user of the hook overrides the hook name to be `foo`, then the `defaults` object will provide a default value for `sails.config.foo.name`.

Using `defaults` as a function

If you specify a function for the `defaults` feature instead of a plain object, it takes a single argument (`config`) which receives any Sails configuration overrides. Configuration overrides can be made by passing settings to the command line when lifting Sails (e.g. `sails lift --prod`), by passing an object as the first argument when programmatically lifting or loading Sails (e.g. `Sails.lift({port: 1338}, ...)`) or by using a `.sailsrc` file. The `defaults` function should return a plain object representing configuration defaults for your hook.

`.initialize(cb)`

The `initialize` feature allows a hook to perform startup tasks that may be asynchronous or rely on other hooks. All Sails configuration is guaranteed to be completed before a hook's `initialize` function runs. Examples of tasks that you may want to put in `initialize` are:

- Logging in to a remote API
- Reading from a database that will be used by hook methods
- Loading support files from a user-configured directory
- Waiting for another hook to load first

Like all hook features, `initialize` is optional and can be left out of your hook definition. If implemented, `initialize` takes one argument: a callback function which must be called in order for Sails to finish loading:

```
initialize: function(cb) {

   // Do some stuff here to initialize hook
   // And then call `cb` to continue
   return cb();

}
```

Hook timeout settings

By default, hooks have ten seconds to complete their `initialize` function and call `cb` before Sails throws an error. That timeout can be configured by setting the `_hookTimeout` key to the number of milliseconds that Sails should wait. This can be done in the hook's `defaults`:

```
defaults: {
   __configKey__: {
      _hookTimeout: 20000 // wait 20 seconds before timing out
   }
}
```

Hook events and dependencies

When a hook successfully initializes, it emits an event with the following name:

`hook:<hook name>:loaded`

For example:

- The core `orm` hook emits `hook:orm:loaded` after its initialization is complete.

- A hook installed into `node_modules/sails-hook-foo` emits `hook:foo:loaded` by default
- The same `sails-hook-foo` hook, with `sails.config.installedHooks['sails-hook-foo'].name` set to `bar` would emit `hook:bar:loaded`
- A hook installed into `node_modules/mygreathook` would emit `hook:mygreathook:loaded`
- A hook installed into `api/hooks/mygreathook` would also emit `hook:mygreathook:loaded`

You can use the "hook loaded" events to make one hook dependent on another. To do so, simply wrap your hook's `initialize` logic in a call to `sails.on()`. For example, to make your hook wait for the `orm` hook to load, you could make your `initialize` similar to the following:

```
initialize: function(cb) {

   sails.on('hook:orm:loaded', function() {

      // Finish initializing custom hook
      // Then call cb()
      return cb();

   }
}
```

To make a hook dependent on several others, gather the event names to wait for into an array and call `sails.after`:

```
initialize: function(cb) {

   var eventsToWaitFor = ['hook:orm:loaded', 'hook:mygreathook:loaded'];
   sails.after(eventsToWaitFor, function() {

      // Finish initializing custom hook
      // Then call cb()
      return cb();

   }
}
```

`.routes`

The `routes` feature allows a custom hook to easily bind new routes to a Sails app at load time. If implemented, `routes` should be an object with either a `before` key, an `after` key, or both. The values of those keys should in turn be objects whose keys are route addresses, and whose values are route-handling functions with the standard `(req, res, next)` parameters. Any routes specified in the `before` object will be bound *before* custom user routes (as defined in sails.config.routes) and blueprint routes. Conversely, routes specified in the `after` object will be bound *after* custom and blueprint routes. For example, consider the following `count-requests` hook:

```
module.exports = function (sails) {

   return {

      initialize: function(cb) {
         this.numRequestsSeen = 0;
         this.numUnhandledRequestsSeen = 0;
         return cb();
      },

      routes: {
         before: {
            'GET /*': function (req, res, next) {
               this.numRequestsSeen++;
               return next();
            }
         },
         after: {
            'GET /*': function (req, res, next) {
               this.numUnhandledRequestsSeen++;
               return next();
            }
         }
      };
};
```

This hook will process all requests via the function provided in the `before` object, and increment its `numRequestsSeen` variable. It will also process any *unhandled* requests via the function provided in the `after` object—that is, any routes that aren't bound in the app via a custom route configuration or a blueprint.

> The two variables set up in the hook will be available to other modules in the Sails app as `sails.hooks["count-requests"].numRequestsSeen` and `sails.hooks["count-requests"].numUnhandledRequestsSeen`

Creating an Installable Hook

Installable hooks are custom Sails hooks that reside in an application's `node_modules` folder. They are useful when you want to share functionality between Sails apps, or publish your hook to NPM to share it with the Sails community. If you wish to create a hook for use in *just one* Sails app, see creating a project hook instead.

To create a new installable hook:

1. Choose a name for your new hook. It must not conflict with any of the core hook names.
2. Create a new folder on your system with the name `sails-hook-<your hook name>`. The `sails-hook-` prefix is optional but recommended for consistency; it is stripped off by Sails when the hook is loaded.
3. Create a `package.json` file in the folder. If you have `npm` installed on your system, you can do this easily by running `npm init` and following the prompts. Otherwise, you can create the file manually, and ensure that it contains at a minimum the following:

   ```
   {
    "name": "sails-hook-your-hook-name",
    "version": "0.0.0",
    "description": "a brief description of your hook",
    "main": "index.js",
    "sails": {
      "isHook": true
    }
   }
   ```

   If you use `npm init` to create your `package.json`, be sure to open the file afterwards and manually insert the `sails` key containing `isHook: true`.
4. Write your hook code in `index.js` in accordance with the hook specification.

Your new folder may contain other files as well, which can be loaded in your hook via `require`; only `index.js` will be read automatically by Sails. Use the `dependencies` key of your `package.json` to refer to any dependencies that need to be installed in order for your hook to work (you may also use `npm install <dependency> --save` to easily save dependency information to `package.json`).

Testing your new hook

Before you distribute your installable hook to others, you'll want to write some tests for it. This will help ensure compatibility with future Sails versions and significantly reduce hair-pulling and destruction of nearby objects in fits of rage. While a full guide to writing tests is

outside the scope of this doc, the following steps should help get you started:

1. Add Sails as a `devDependency` in your hook's `package.json` file:

```
"devDependencies": {
   "sails": "~0.11.0"
}
```

2. Install Sails as a dependency of your hook with `npm install sails` or `npm link sails` (if you have Sails installed globally on your system).
3. Install Mocha on your system with `npm install -g mocha`, if you haven't already.
4. Add a `test` folder inside your hook's main folder.
5. Add a `basic.js` file with the following basic test:

```
var Sails = require('sails').Sails;

describe('Basic tests ::', function() {

    // Var to hold a running sails app instance
    var sails;

    // Before running any tests, attempt to lift Sails
    before(function (done) {

        // Hook will timeout in 10 seconds
        this.timeout(11000);

        // Attempt to lift sails
        Sails().lift({
          hooks: {
            // Load the hook
            "your-hook-name": require('../'),
            // Skip grunt (unless your hook uses it)
            "grunt": false
          },
          log: {level: "error"}
        },function (err, _sails) {
          if (err) return done(err);
          sails = _sails;
          return done();
        });
    });

    // After tests are complete, lower Sails
    after(function (done) {

        // Lower Sails (if it successfully lifted)
        if (sails) {
            return sails.lower(done);
        }
        // Otherwise just return
        return done();
    });

    // Test that Sails can lift with the hook in place
    it ('sails does not crash', function() {
        return true;
    });

});
```

6. Run the test with `mocha -R spec` to see full results.
7. See the Mocha docs for a full reference.

Publishing your hook

Assuming your hook is tested and looks good, and assuming that the hook name isn’t already in use by another NPM module, you can share it with world by running `npm publish` . Go you!

- Hooks overview
- Using hooks in your app
- The hook specification
- Creating a project hook

Creating a Project Hook

Project hooks are custom Sails hooks that reside in an application's `api/hooks` folder. They are typically useful when you want to take advantage of hook features like defaults and routes for code that is used by multiple components in a single app. If you wish to re-use a hook in *more than one* Sails app, see creating an installable hook instead.

To create a new project hook:

1. Choose a name for your new hook. It must not conflict with any of the core hook names.
2. Create a folder with that name in your app's `api/hooks` folder.
3. Add an `index.js` file to that folder.
4. Write your hook code in `index.js` in accordance with the hook specification.

Your new folder may contain other files as well, which can be loaded in your hook via `require`; only `index.js` will be read automatically by Sails.

As an alternative to a folder, you may create a file in your app's `api/hooks` folder like `api/hooks/myProjectHook.js`.

Testing that your hook loads properly

To test that your hook is being loaded by Sails, lift your app with `sails lift --verbose`. If your hook is loaded, you will see a message like:

`verbose: your-hook-name hook loaded successfully.`

in the logs.

- Hooks overview
- Using hooks in your app
- The hook specification
- Creating an installable hook

Using Hooks in a Sails App

Using a project hook

To use a project hook in your app, first create the `api/hooks` folder if it doesn't already exist. Then create the project hook or copy the folder for the hook you want to use into `api/hooks`.

Using an installable hook

To use an installable hook in your app, simply run `npm install` with the package name of the hook you wish to install (e.g. `npm install sails-hook-autoreload`). You may also manually copy or link an installable hook folder that you've created directly into your app's `node_modules` folder.

Calling hook methods

Any methods that a hook exposes are available in the `sails.hooks[<hook-name>]` object. For example, the `sails-hook-email` hook provides a `sails.hooks.email.send()` method (note that the `sails-hook-` prefix is stripped off). Consult a hook's documentation to determine which methods it provides.

Configuring a hook

Once you've added an installable hook to your app, you can configure it using the regular Sails config files like `config/local.js`, `config/env/development.js`, or a custom config file you create yourself. Hook settings are typically namespaced under the hook's name, with any `sails-hook-` prefix stripped off. For example, the `from` setting for `sails-hook-email` is available as `sails.config.email.from`. The documentation for the installable hook should describe the available configuration options.

Changing the way Sails loads an installable hook

On rare occassions, you may need to change the name that Sails uses for an installable hook, or change the configuration key that the hook uses. This may be the case if you already have a project hook with the same name as an installable hook, or if you’re already using a configuration key for something else. To avoid these conflicts, Sails provides the `sails.config.installedHooks.<hook-identity>` configuration option. The hook identity is *always* the name of the folder that the hook is installed in.

```
// config/installedHooks.js
module.exports.installedHooks = {
   "sails-hook-email": {
      // load the hook into sails.hooks.emailHook instead of sails.hooks.email
      "name": "emailHook",
      // configure the hook using sails.config.emailSettings instead of sails.config.emai
      "configKey": "emailSettings"
   }
};
```

> Note: you may have to create the `config/installedHooks.js` file yourself.

- Hooks overview
- The hook specification
- Creating a project hook
- Creating an installable hook

Getting Started

Installation

To install the latest stable release with the command-line tool:

```
sudo npm -g install sails
```

On Windows (or Mac OS with Homebrew), you don't need sudo:

```
npm -g install sails
```

Creating a New Sails Project

Create a new app:

```
sails new testProject
```

Now lift the server:

```
cd testProject
sails lift
```

At this point, if you visit (http://localhost:1337/) you will see the default home page.

Now, let's get Sails to do cool stuff.

New to Node.js?

That's okay! We'll get you pointed in the right direction.

Per nodejs.org:

> "Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."

More simply put, Node.js allows us to quickly and efficiently run JavaScript code outside the browser, making it possible to use the same language on both the frontend and the backend.

What OS do I need?

Node.js will install on most major Operating systems. MacOSX, many flavors of Linux, and Windows are supported.

Now, lets take a look at what OS you have. Please choose from the following for instructions on setting up Node.js:

I have Mac OSX

I have Linux

I have Windows

Install on OSX

Using a package:

Simply download Macintosh Installer.

Using homebrew:

```
brew install node
```

Using macports:

```
port install nodejs
```

Install on Linux

Ubuntu, Mint

Example install:

```
sudo apt-get install python-software-properties python g++ make
curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -
sudo apt-get install -y nodejs
```

It installs current stable Node on the current stable Ubuntu. Quantal (12.10) users may need to install the *software-properties-common* package for the `add-apt-repository` command to work: `sudo apt-get install software-properties-common`

There is a naming conflict with the node package (Amateur Packet Radio Node Program), and the nodejs binary has been renamed from `node` to `nodejs`. You'll need to symlink `/usr/bin/node` to `/usr/bin/nodejs` or you could uninstall the Amateur Packet Radio Node Program to avoid that conflict.

Fedora

Node.js and npm are available in Fedora 18 and later. Just use your favorite graphical package manager or run this on a terminal to install both npm and node:

```
sudo yum install npm
```

RHEL/CentOS/Scientific Linux 6

Node.js and npm are available from the Fedora Extra Packages for Enterprise Linux (EPEL) *testing* repository. If you haven't already done so, first enable EPEL and then run the following command to install node and npm:

```
su -c 'yum --enablerepo=epel-testing install npm'
```

Arch Linux

Node.js is available in the Community Repository.

```
pacman -S nodejs
```

Gentoo

Node.js is available in official gentoo portage tree. You have to unmask it.

```
# emerge -aqv --autounmask-write nodejs
# etc-update
# emerge -aqv nodejs
```

Debian, LMDE

For *Debian sid (unstable)*, Node.js is available in the official repo.

For *Debian Wheezy (stable)*, Node.js is available in wheezy-backports. To install backports, add this to your sources.list (`/etc/apt/sources.list`):

```
deb http://YOURMIRROR.debian.org/debian wheezy-backports main
```

Then run:

```
apt-get update
apt-get install nodejs
```

For *Debian Squeeze (oldstable)*, your best bet is to compile node by yourself (as `root`):

```
apt-get install python g++ make
mkdir ~/nodejs && cd $_
wget -N http://nodejs.org/dist/node-latest.tar.gz
tar xzvf node-latest.tar.gz && cd `ls -rd node-v*`
./configure
make install
```

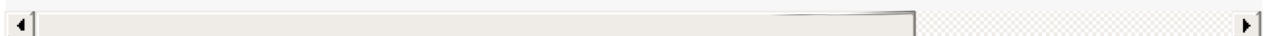
openSUSE & SLE

Node.js stable repos list. Also node.js is available in openSUSE:Factory repository.

Available RPM packages for: openSUSE 11.4, 12.1, Factory and Tumbleweed; SLE 11 (with SP1 and SP2 variations).

Example install on openSUSE 12.1:

```
sudo zypper ar http://download.opensuse.org/repositories/devel:/languages:/nodejs/openSUS
sudo zypper in nodejs nodejs-devel
```



FreeBSD and OpenBSD

Node.js is available through the ports system.

```
/usr/ports/www/node
```

Development versions are also available using ports

```
cd /usr/ports/www/node-devel/ && make install clean
```

or packages on FreeBSD

```
pkg_add -r node-devel
```

The Node Package Manager is not installed along with Node.js by default on FreeBSD and will be needed for development and installing dependencies.

```
/usr/ports/www/npm
```

Also note that FreeBSD 10 using clang will conflict with the occasional build scrpt (which assumes gcc) using node-gyp, and can be resolved by setting an envvar.

```
CXX=c++
```

Install on Windows

Using a package:

Simply download Windows Installer.

Using chocolatey to install Node:

```
cinst nodejs
```

or for full install with NPM:

```
cinst nodejs.install
```

On to Sails.js!

Once Node.js is installed on your system, you can go ahead and install Sails.

Further help!

We know that sometimes things don't go as planned. If you still have any issue with this, please feel free to visit Node.js's IRC Channel or our own IRC Channel.

Sails是什么？

当然，`Sails` 是一个 `web框架` 。但退一步，这又是什么意思呢？有时，当我们提到 `web` 的时候，我们指的是"front-end web"(web前端）。 我们想到的是Web标准的概念，像HTML5或CSS3；以及框架， `Backbone` 、 `Angular` , 或 `jQuery` 。 `sails` 可不是“这种”的 `web` 框架。 `Sails` 可以与 `Angular` 和 `Backbone` 工作的很好，但绝不会使用 `Sails` 取代这些包。

换句话说，当我们谈论 `Web框架` 的时候，我们指的是“back-end web”（服务器端）。这里提到的概念是 `REST` 、 `HTTP` ,或 `WebSockets` ， 以及 `Java` 、 `Ruby` 、 `Node.js` 等技术。一个“back-end web”框架有助于你构建API，处理HTML文件成千上万的并发用户。 `Sails` 就是 “这种” `web框架` 。

约定优于配置

Sails accomplishes many of the same goals as other MVC web application frameworks, using many of the same methodologies. This was done on purpose. A consistent approach makes developing apps more predictable and efficient for everybody involved.

Imagine starting a new job at a company building a Sails app (or imagine starting the company, if that's your thing.) If anyone on your team has worked with frameworks like Zend, Laravel, CodeIgniter, Cake, Grails, Django, ASP.NET MVC, or Rails, Sails will feel pretty familiar. Not only that, but they can look at a Sails project and know, generally, how to code up the basic patterns they've implemented over and over again in the past; whether their background is in PHP, Ruby, Java, C#, or Node.js. What about your second app, or your third? Each time you create a new Sails app, you start with a sane, familiar boilerplate that makes you more productive. In many cases, you'll even be able to recycle some of your backend code.

> **History**
>
> Sails didn't invent this concept-- it's been around for years. Even before the phrase "Convention over Configuration" (or CoC) was popularized by Ruby on Rails, it was a core tenet of the JavaBeans specification and in many ways, a natural lashback against the extremely verbose XML configuration common in traditional Java web frameworks of the late '90s and early 2000s.

松耦合

The days of forcing a one-size-fits-all approach to development are over. We need tools that allow us to pick and choose the components that fit our requirements. In fact, it’s just plain lazy to create things any other way. Sails’s approach is to loosely couple components so that they can be added or subtracted from your app at will.

Node at its core has created a “can do” culture eager to experiment and make things work. Sails embraces this attitude and strives to deliver tools that work around you. The level of automation or magic you want in Sails is directly tied to the time you have for a project and your experience working with Node. Sails is flexible enough to allow you to explore and create when you have the time but also provides automation when you don’t.

Sails accomplishes this loose coupling using plain-old require. No magic, other than the time to craft components that can be part of the whole but don’t need to be present in order for the whole to work. For example, controllers, models, and configuration files are just Node modules. Sails uses some convention to help. Sails picks up on the name UserController.js in the Controllers folder to deduce that this is indeed a user controller. Another example involves policies. So policies allow you to have a bit of code that executes on controller or specific controller action. The cool part is that the configuration file that connects the policy with the controller/action are separate. That means you can write a bunch of different policies and they are completely portable between Sails apps. You can decide later which controller/actions you want to apply them to.

Sails core consists of twenty different hooks: modules that modify the server runtime, adding middleware, binding route listeners, or otherwise attaching additional capabilities to the framework. This gives you access to override or disable every component and configuration parameter in Sails. These hooks are loaded at run-time when Sails starts. You even have the ability to have one-time configuration for your hook itself. This is actually one of the key differentiators between hooks and services.

Another example of loose coupling is configuration files. Need some configuration to be available for your project? No problem. Create a file in the config folder that uses the common module.exports pattern and everything in that module is available for you from the sails global object.

Almost every component of Sails can either be omitted, overwritten, or extended. For example, Sails has a group of tools called blueprints. These blueprints make it really easy to get a project up and running with regard to routes and CRUD operations. But suppose you want to use the read, update, and delete operations but the create action needs some tender loving care. No problem, just build a create action and the other CRUD operations keep working. Your custom action subs in for the blueprint action. It’s just that simple.

> Links:
>
> - Unix philosophy
> - Node culture

sails-docs-reference

Reference Section of Sails.js documentation

ideas for some restructuring:

```
|
|- Usage
  |
  |- Request (req)
  |
  |- Response (res)
  |
  |- Config (sails.config)
  |
  |- Model (sails.models)
  |
  |- Sockets (sails.sockets)
  |
  |- Sails CLI
  |
  |- Blueprint API
  |
  |- Browser SDK (sails.io.js)

|
|- Concepts
   |
   |- Security
   |
   |- Deployment
      |
      |- FAQ
      |- Hosting
      |- Scaling
   |
   |- Testing
   |
   |- Internationalization
   |
   |- Logging
   |
   |- File Uploads
   |
   |- Assets & Tasks
   |
   |- Middleware
```

```
      |
      |- Routes
      |
      |- Blueprints
      |
      |- Policies
      |
      |- Controllers
      |
      |- Custom Responses
      |
      |- Models
         |
         |- Attributes
         |- Validations
         |- Associations
         |- Lifecycle Callbacks
         |- Custom Schemas
         |- Migrations
      |
      |- Services
         |- todo: add example: Sending Email
      |
      |- Views

   |
   |- Advanced
      |
      |- Plugins
      |
      |- Programmatic Usage
      |
      |- Globals
      |
      |- .sailsrc
```

Blueprint API

Overview

Together, blueprint routes and blueprint actions constitute the **blueprint API**, the built-in logic that powers the RESTful JSON API you get every time you create a model and controller.

For example, if you create a `User.js` model and `UserController.js` controller file in your project, then with blueprints enabled you will be able to immediately visit `/user/create?name=joe` to create a user, and visit `/user` to see an array of your app's users. All without writing a single line of code!

Blueprints are great for prototyping, but they are also a powerful tool in production due to their ability to be overridden, protected, extended or disabled entirely.

Blueprint Routes

When you run `sails lift` with blueprints enabled, the framework inspects your controllers, models, and configuration in order to bind certain routes automatically. These implicit blueprint routes (sometimes called "shadows") allow your app to respond to certain requests without you having to bind those routes manually in your `config/routes.js` file. By default, the blueprint routes point to their corresponding blueprint *actions* (see "Blueprint Actions" below), any of which can be overridden with custom code.

There are three types of blueprint routes in Sails:

- **RESTful routes**, where the path is always `/:modelIdentity` or `/:modelIdentity/:id`. These routes use the HTTP "verb" to determine the action to take; for example a `POST` request to `/user` will create a new user, and a `DELETE` request to `/user/123` will delete the user whose primary key is 123. In a production environment, RESTful routes should generally be protected by policies to avoid unauthorized access.
- **Shortcut routes**, where the action to take is encoded in the path. For example, the `/user/create?name=joe` shortcut creates a new user, while `/user/update/1?name=mike` updates user #1. These routes only respond to `GET` requests. Shortcut routes are very handy for development, but generally should be disabled in a production environment.
- **Action routes**, which automatically create routes for your custom controller actions. For example, if you have a `FooController.js` file with a `bar` method, then a `/foo/bar` route will automatically be created for you as long as blueprint action routes are enabled. Unlike RESTful and shortcut routes, action routes do *not* require that a

controller has a corresponding model file.
See the blueprints subsection of the configuration reference for blueprint configuration options, including how to enable / disable different blueprint route types.

Blueprint Actions

Blueprint actions (not to be confused with blueprint action *routes*) are generic actions designed to work with any of your controllers that have a model of the same name (e.g. `ParrotController` would need a `Parrot` model). Think of them as the default behavior for your application. For instance, if you have a `User.js` model and an empty `UserController.js` controller, `find` , `create` , `update` , `destroy` , `populate` , `add` and `remove` actions exist implicitly, without you having to write them.

By default, the blueprint RESTful routes and shortcut routes are bound to their corresponding blueprint actions. However, any blueprint action can be overridden for a particular controller by creating a custom action in that controller file (e.g. `ParrotController.find`). Alternatively, you can override the blueprint action *everywhere in your app* by creating your own custom blueprint action. (e.g. `api/blueprints/create.js`).

The current version of Sails ships with the following blueprint actions:

- find
- findOne
- create
- update
- destroy
- populate
- add
- remove

Consequently, the blueprint API methods covered in this section of the documentation correspond one-to-one with the blueprint actions above.

Overriding Blueprints

(taken from https://stackoverflow.com/questions/22273789/crud-blueprint-overriding-in-sailsjs)

To override blueprints in Sails v0.10, you create an api/blueprints folder and add your blueprint files (e.g. find.js, create.js, etc.) within. You can take a look at the code for the default actions in the Sails blueprints hook for a head start.

Note: Currently all files must be lowercase! (The default actions contains findOne.js, but in /api/blueprints it needs to be findone.js)

Adding custom blueprints is also supported, but they currently do not get bound to routes automatically. If you create a /blueprints/foo.js file, you can bind a route to it in your /config/routes.js file with (for example):

```
GET /myRoute: {blueprint: 'foo'}
```

Disabling blueprints on a per-controller basis

You may also override any of the settings from `config/blueprints.js` on a per-controller basis by defining a '_config' key in your controller definition, and assigning it a configuration object with overrides for the settings in this file.

```
module.exports = {
  _config: {
    actions: false,
    shortcuts: false,
    rest: false
  }
}
```

Notes

> - While the following documentation focuses on HTTP, the blueprint API (just like any of your custom actions and policies) is also compatible with WebSockets, thanks to the request interpreter. Check out the reference section on the browser SDK for example usage.

Add to Collection

Adds an association between two records.

```
POST /:model/:record/:association/:record_to_add?
```

This action pushes a reference to some other record (the "foreign" record) onto a collection attribute of this record (the "primary" record).

- If `:record_to_add` of an existing record is supplied, it will be associated with the primary record.
- If no `:record_to_add` is supplied, and the body of the **POST** contains values for a new record, that record will be created and associated with the primary record.
- If the collection within the primary record already contains a reference to the foreign record, this action will be ignored.
- If the association is 2-way (i.e. reflexive, with "via" on both sides) the association on the foreign record will also be updated.

Example

Add purchase 47 to the list of purchases that Dolly (employee #7) has been involved in.

Using jQuery:

```
$.post('/employee/7/involvedInPurchases/47', function (purchases) {
  console.log(purchases);
});
```

Using Angular:

```
$http.post('/employee/7/involvedInPurchases/47')
.then(function (purchases) {
  console.log(purchases);
});
```

Using sails.io.js:

```
io.socket.post('/employee/7/involvedInPurchases/47', function (purchases) {
  console.log(purchases);
});
```

Using cURL:

```
curl http://localhost:1337/employee/7/involvedInPurchases/47 -X "POST"
```

Should return "Dolly", the primary record:

```
{
  "involvedInPurchases": [
    {
      "amount": 10000,
      "createdAt": "2014-08-03T01:50:33.898Z",
      "updatedAt": "2014-08-03T01:51:08.227Z",
      "id": 47,
      "cashier": 7
    }
  ],
  "name": "Dolly",
  "createdAt": "2014-08-03T01:16:35.440Z",
  "updatedAt": "2014-08-03T01:51:41.567Z",
  "id": 7
}
```

Notes

> - This action is for dealing with *plural* ("collection") associations. If you want to set or unset a *singular* ("model") association, just use update.
> - The example above assumes "rest" blueprints are enabled, and that your project contains at least an 'Employee' model with association: `involvedInPurchases: {collection: 'Purchase', via: 'cashier'}` as well as a `Purchase` model with association: `cashier: {model: 'Employee'}`. You'll also need at least an empty `PurchaseController` and `EmployeeController`. You can quickly achieve this by running:
>
> ```
> $ sails new foo
> $ cd foo
> $ sails generate api purchase
> $ sails generate api employee
> ```
>
> ...then editing `api/models/Purchase.js` and `api/models/Employee.js`.

Create a Record

Creates a new model instance in your database then returns it's values.

```
POST /:model
```

Attributes can be sent in the HTTP body as form-encoded values or JSON.

Responds with a JSON object representing the newly created instance. If a validation error occurred, a JSON response with the invalid attributes and a `400` status code will be returned instead.

Additionally, a `create` event will be published to all listening sockets (see the docs for .watch() for more info).

If the action is triggered via a socket request, the requesting socket will ALSO be subscribed to the newly created model instance. If the record is subsequently updated or deleted, a message will be sent to that socket's client informing them of the change. See the docs for .subscribe() for more info.

Parameters

Parameter	Type	Details
*	((string)) ((number)) ((object)) ((array))	For `POST` (RESTful) requests, pass in body parameter with the same name as the attribute defined on your model to set those values on your new record. For `GET` (shortcut) requests, add the parameters to the query string. Nested objects and arrays passed in as parameters are handled the same way as if they were passed into the model's .create() method.
callback	((string))	If specified, a JSONP response will be sent (instead of JSON). This is the name of the client-side javascript function to call, passing results as the first (and only) argument e.g. `?callback=myJSONPHandlerFn`

Examples

Create a record (REST)

Create a new pony named "AppleJack" with a hobby of "pickin".

Route

`POST /pony`

JSON Request Body

```
{
  "name": "AppleJack",
  "hobby": "pickin"
}
```

Example Response

```
{
  "name": "AppleJack",
  "hobby": "pickin",
  "id": 47,
  "createdAt": "2013-10-18T01:23:56.000Z",
  "updatedAt": "2013-11-26T22:55:19.951Z"
}
```

Create a record (shortcuts)

Route

`GET /pony/create?name=Fluttershy&best_pony=yep`

Expected Response

```
{
 "name": "Fluttershy",
 "best_pony": "yep",
 "createdAt": "2014-02-24T21:02:16.068Z",
 "updatedAt": "2014-02-24T21:02:16.068Z",
 "id": 5
}
```

Examples with One Way Associations

You can create associations between models in two different ways. You can either make the association with a record that already exists OR you can create both records simultaneously. Check out the examples to see how.

These examples assume the existence of `Pet` and `Pony` APIs which can be created by hand or using the Sails CLI Tool. The `Pony` model must be configured with a `pet` attribute pointing to the `Pet` model. See Model Association Docs for info on how to do this.

Create record while associating w/ existing record (REST)

Create a new pony named "Pinkie Pie" and associate it with an existing pet named "Gummy" which has an `id` of 10.

Route

`POST /pony`

JSON Request Body

```
{
  "name": "Pinkie Pie",
  "hobby": "ice skating",
  "pet": 10
}
```

Example Response

```
{
  "name": "Pinkie Pie",
  "hobby": "ice skating",
  "pet": {
    "name": "Gummy",
    "species": "crocodile",
    "id": 10
  },
  "id": 4,
  "createdAt": "2013-10-18T01:22:56.000Z",
  "updatedAt": "2013-11-26T22:54:19.951Z"
}
```

Create new record while associating w/ another new record (REST)

Create a new pony named "Pinkie Pie", an "ice skating" hobby, and a new pet named "Gummy".

Route

`POST /pony`

JSON Request Body

```
{
  "name": "Pinkie Pie",
  "hobby": "ice skating",
  "pet": {
    "name": "Gummy",
    "species": "crocodile"
  }
}
```

Expected Response

```
{
  "name": "Pinkie Pie",
  "hobby": "ice skating",
  "pet": {
    "name": "Gummy",
    "species": "crocodile",
    "id": 10
  },
  "id": 4,
  "createdAt": "2013-10-18T01:22:56.000Z",
  "updatedAt": "2013-11-26T22:54:19.951Z"
}
```

Destroy a Record

Deletes an existing record specified by `id` from the database forever and returns the values of the deleted record.

```
DELETE /:model/:record
```

Destroys the model instance which matches the **id** parameter. Responds with a JSON object representing the newly destroyed instance. If no model instance exists matching the specified **id**, a `404` is returned.

Additionally, a `destroy` event will be published to all sockets subscribed to the instance room.

Consequently, all sockets currently subscribed to the instance will be unsubscribed from it.

Parameters

Parameter	Type	Details
id *(required)*	((number)) *-or-* ((string))	The primary key value of the record to destroy. For `POST` (RESTful) requests, this can be supplied in the JSON body or as part of the route path. For `GET` (shortcut) requests, it must be supplied in the route path.
callback	((string))	If specified, a JSONP response will be sent (instead of JSON). This is the name of the client-side javascript function to call, passing results as the first (and only) argument e.g. `?callback=myJSONPHandlerFn`

Examples

Destroy (REST)

Delete Pinkie Pie.

Route

`DELETE /pony`

JSON Request Body

```
{
  "id": 4
}
```

Expected Response

```
{
  "name": "Pinkie Pie",
  "hobby": "kickin",
  "id": 4,
  "createdAt": "2013-10-18T01:23:56.000Z",
  "updatedAt": "2013-11-26T22:55:19.951Z"
}
```

Destroy (Shortcuts)

Route

`GET /pony/destroy/4`

Expected Response

Same as above.

Find Records

Returns a list of records from the model as a JSON array of objects.

```
GET /:model
```

Results may be filtered, paginated, and sorted based on the blueprint configuration and/or parameters sent in the request.

If the action was triggered via a socket request, the requesting socket will be "subscribed" to all records returned. If any of the returned records are subsequently updated or deleted, a message will be sent to that socket's client informing them of the change. See the docs for Model.subscribe() for details.

Parameters

All parameters are optional.

Parameter	Type	Details
*	((string))	To filter results based on a particular attribute, specify a query parameter with the same name as the attribute defined on your model. For instance, if our `Purchase` model has an **amount** attribute, we could send `GET /purchase?amount=99.99` to return a list of $99.99 purchases.
where	((string))	Instead of filtering based on a specific attribute, you may instead choose to provide a `where` parameter with a Waterline WHERE criteria object, *encoded as a JSON string*. This allows you to take advantage of `contains` , `startsWith` , and other sub-attribute criteria modifiers for more powerful `find()` queries. e.g. `?where={"name":{"contains":"theodore"}}`
limit	((number))	The maximum number of records to send back (useful for pagination). Defaults to 30. e.g. `?limit=100`
skip	((number))	The number of records to skip (useful for pagination). e.g. `?skip=30`
sort	((string))	The sort order. By default, returned records are sorted by primary key value in ascending order. e.g. `?sort=lastName%20ASC`
populate	((string))	If specified, overide the default automatic population process. Accepts a comma separated list of attributes names for which to populate record values. See here for more information on how the population process fills out attributes in the returned list of records according to the model's defined associations.
callback	((string))	If specified, a JSONP response will be sent (instead of JSON). This is the name of a client-side javascript function to call, to which results will be passed as the first (and only) argument e.g. ?callback=my_JSONP_data_receiver_fn

`find` Example

Find the 30 newest purchases in our database.

```
[
 {
   "amount": 49.99,
   "id": 1,
   "createdAt": "2013-10-18T01:22:56.000Z",
   "updatedAt": "2013-10-18T01:22:56.000Z"
 },
 {
   "amount": 99.99,
   "id": 47,
   "createdAt": "2013-10-14T01:22:00.000Z",
   "updatedAt": "2013-10-15T01:20:54.000Z"
 }
]
```

Using jQuery:

```
$.get('/purchase?sort=createdAt DESC', function (purchases) {
  console.log(purchases);
});
```

Using Angular:

```
$http.get('/purchase?sort=createdAt DESC')
.then(function (res) {
  var purchases = res.data;
  console.log(purchases);
});
```

Using sails.io.js:

```
io.socket.get('/purchase?sort=createdAt DESC', function (purchases) {
  console.log(purchases);
});
```

Using cURL:

```
curl http://localhost:1337/purchase?sort=createdAt%20DESC
```

Notes

> - The example above assumes "rest" blueprints are enabled, and that your project contains a `Purchase` model and an empty `PurchaseController`. You can quickly achieve this by running:
>
> ```
> $ sails new foo
> $ cd foo
> $ sails generate api purchase
> ```

Find One

Returns a single record from the model as a JSON Object.

```
GET /:model/:id
```

The **findOne()** blueprint action returns a single record from the model (given by `:modelIdentity`) as a JSON object. The specified `id` is the primary key of the desired record.

If the action was triggered via a socket request, the requesting socket will be "subscribed" to the returned record. If the record is subsequently updated or deleted, a message will be sent to that socket's client informing them of the change. See the docs for .subscribe() for more info.

Parameters

Parameter	Type	Details
id *(required)*	((number)) *-or-* ((string))	The desired record's primary key value e.g. `/product/7`
callback	((string))	If specified, a JSONP response will be sent (instead of JSON). This is the name of the client-side javascript function to call, passing results as the first (and only) argument e.g. `?callback=myJSONPHandlerFn`

Example

Find the purchase with ID #1, E.g. `http://localhost:1337/purchase/1`

Route

`GET /purchase/1`

Expected Response

```
{
  "amount": 49.99,
  "id": 1,
  "createdAt": "2013-10-18T01:22:56.000Z",
  "updatedAt": "2013-10-18T01:22:56.000Z"
}
```

Populate Where...

If the specified association is plural ("collection"), this action returns the list of associated records as a JSON array of objects. If the specified association is singular ("model"), this action returns the associated record as a JSON object.

```
GET /:model/:record/:association
```

Example

Populate the `cashier` who conducted purchase #47.

Using jQuery:

```
$.get('/purchase/47/cashier', function (purchase) {
  console.log(purchase);
});
```

Using Angular:

```
$http.get('/purchase/47/cashier')
.then(function (purchase) {
  console.log(purchase);
});
```

Using sails.io.js:

```
io.socket.get('/purchase/47/cashier', function (purchase) {
  console.log(purchase);
});
```

Using cURL:

```
curl http://localhost:1337/purchase/47/cashier
```

Should return:

```
{
  "amount": 99.99,
  "id": 47,
  "cashier": {
    "name": "Dolly",
    "id": 7,
    "createdAt": "2012-05-14T01:21:05.000Z",
    "updatedAt": "2013-01-15T01:18:40.000Z"
  },
  "createdAt": "2013-10-14T01:22:00.000Z",
  "updatedAt": "2013-10-15T01:20:54.000Z"
}
```

Notes

> - The example above assumes "rest" blueprints are enabled, and that your project contains at least an empty 'Employee' model as well as a `Purchase` model with an association attribute: `cashier: {model: 'Employee'}`. You'll also need at least an empty `PurchaseController` and `EmployeeController`. You can quickly achieve this by running:
>
>   ```
>   $ sails new foo
>   $ cd foo
>   $ sails generate api purchase
>   $ sails generate api employee
>   ```
>
>   ...then editing `api/models/Purchase.js`.

Remove from Collection

Removes an association between two records.

```
DELETE /:model/:record/:association/:record_to_remove
```

This action removes a reference to some other record (the "foreign" record) from a collection attribute of this record (the "primary" record).

- If the foreign record does not exist, it is created first.
- If the collection doesn't contain a reference to the foreign record, this action will be ignored.
- If the association is 2-way (i.e. reflexive, with "via" on both sides) the association on the foreign record will also be updated.

Example

Remove Dolly (employee #7) from the `employeesOfTheMonth` list of store #16.

Using jQuery:

```
$.delete('/store/16/employeesOfTheMonth/7', function (purchases) {
  console.log(purchases);
});
```

Using Angular:

```
$http.delete('/store/16/employeesOfTheMonth/7')
.then(function (purchases) {
  console.log(purchases);
});
```

Using sails.io.js:

```
io.socket.delete('/store/16/employeesOfTheMonth/7', function (purchases) {
  console.log(purchases);
});
```

Using cURL:

```
curl http://localhost:1337/store/16/employeesOfTheMonth/7 -X "DELETE"
```

Should return store #16, the primary record:

```
{
  "employeesOfTheMonth": [],
  "name": "Dolly",
  "createdAt": "2014-08-03T01:16:35.440Z",
  "updatedAt": "2014-08-03T01:51:41.567Z",
  "id": 16
}
```

Notes

> - This action is for dealing with *plural* ("collection") associations. If you want to set or unset a *singular* ("model") association, just use update.
> - The example above assumes "rest" blueprints are enabled, and that your project contains at least an empty 'Employee' model as well as a `Store` model with association: `employeesOfTheMonth: {collection: 'Employee'}`. You'll also need at least an empty `PurchaseController` and `EmployeeController`. You can quickly achieve this by running:
>
> ```
> $ sails new foo
> $ cd foo
> $ sails generate api store
> $ sails generate api employee
> ```
>
> ...then editing `api/models/Store.js`.

Update a Record

Updates an existing record. Attributes to change should be sent in the HTTP body as form-encoded values or JSON.

```
PUT /:model/:record
```

Updates the model instance which matches the **id** parameter. Responds with a JSON object representing the newly updated instance. If a validation error occurred, a JSON response with the invalid attributes and a `400` status code will be returned instead. If no model instance exists matching the specified **id**, a `404` is returned.

Parameters

Parameter	Type	Details
id *(required)*	((number)) *-or-* ((string))	The primary key value of the record to update. e.g. `PUT /product/5`
*	((string)) ((number)) ((object)) ((array))	For `POST` (RESTful) requests, pass in body parameters with the same name as the attributes defined on your model to set those values on the desired record. For `GET` (shortcut) requests, add the parameters to the query string.
callback	((string))	If specified, a JSONP response will be sent (instead of JSON). This is the name of the client-side javascript function to call, passing results as the first (and only) argument e.g. `?callback=myJSONPHandlerFn`

Examples

Update Record (REST)

Change AppleJack's hobby to "kickin".

Route

`PUT /pony/47`

JSON Request Body

```
{
  "hobby": "kickin"
}
```

Expected Response

```
{
  "name": "AppleJack",
  "hobby": "kickin",
  "id": 47,
  "createdAt": "2013-10-18T01:23:56.000Z",
  "updatedAt": "2013-11-26T22:55:19.951Z"
}
```

Update Record (Shortcuts)

`GET /pony/update/47?hobby=kickin`

Expected Response

Same as above.

Add association between two existing records (REST)

Give Pinkie Pie the pre-existing pet named "Bubbles" who has ID 15.

Route

`POST /pony/4/pets`

JSON Request Body

```
{
  "id": 15
}
```

Expected Response

```
{
  "name": "Pinkie Pie",
  "hobby": "kickin",
  "id": 4,
  "pets": [{
      "name": "Gummy",
      "species": "crocodile",
      "id": 10,
      "createdAt": "2014-02-13T00:06:50.603Z",
      "updatedAt": "2014-02-13T00:06:50.603Z"
    },{
      "name": "Bubbles",
      "species": "manticore",
      "id": 15,
      "createdAt": "2014-02-13T00:06:50.603Z",
      "updatedAt": "2014-02-13T00:06:50.603Z"
    }],
  "createdAt": "2013-10-18T01:23:56.000Z",
  "updatedAt": "2013-11-26T22:55:19.951Z"
}
```

Add association between two existing records (Shortcuts)

`GET /pony/4/pets/add/15`

Remove Association (Many-To-Many) (REST)

Remove Pinkie Pie's pet, "Gummy" (ID 12)

Route

`DELETE /pony/4/pets`

JSON Request Body

```
{
  "id": 12
}
```

Expected Response

```
{
  "name": "Pinkie Pie",
  "hobby": "ice skating",
  "pets": [{
      "name": "Bubbles",
      "species": "manticore",
      "id": 15,
      "createdAt": "2014-02-13T00:06:50.603Z",
      "updatedAt": "2014-02-13T00:06:50.603Z"
    }],
  "id": 4,
  "createdAt": "2013-10-18T01:22:56.000Z",
  "updatedAt": "2013-11-26T22:54:19.951Z"
}
```

Remove Association (Many-To-Many) (Shortcuts)

Route

`GET /pony/4/pets/remove/12`

Expected Response

Same as above.

Command Line Interface (CLI)

Overview

Sails comes with a convenient command line tool to quickly get your app scaffolded and running.

sails console

Quietly lift your sails app (i.e. with logging silenced), and enter the node REPL. This means you can access and use all of your models, services, configuration, and much more. Useful for trying out Waterline queries, quickly managing your data, and checking out your project's runtime configuration.

Example

```
$ sails console

info: Starting app in interactive mode...

info: Welcome to the Sails console.
info: ( to exit, type <CTRL>+<C> )

sails>
```

> Note that `sails console` still lifts the server, so your routes will be accessible via HTTP and sockets (e.g. in a browser.)

Global variables in sails console

Sails exposes the same global variables in the console as it does in your app code. This is particularly useful in the REPL. By default, you have access to the `sails` app instance, your models, and your services, as well as Lo-Dash (`sails.util._`) and async (`async`).

> **Warning**
>
> Be careful when using `_` as a variable name in the Node REPL- and when possible, don't. (It doesn't work quite like you'd expect.)
>
> Instead, use lodash as `sails.util._`, e.g.:
>
> ```
> sails> sails.util._.keys(sails.config)
> ```
>
> Or alternatively, build yourself a local variable to use for familiarity:
>
> ```
> sails> var lodash = _;
> ```
>
> Then you can do:
>
> ```
> sails> lodash.keys(sails.config);
> ```

More Examples

Waterline

The format `Model.action(query).exec(console.log)` console.log is good for seeing the results.

```
sails> User.create({name: 'Brian', password: 'sailsRules'}).exec(console.log)
undefined
sails> null { name: 'Brian',
  password: 'sailsRules',
  createdAt: "2014-08-07T04:29:21.447Z",
  updatedAt: "2014-08-07T04:29:21.447Z",
  id: 1 }
```

Pretty cool, it inserts it into the database. However, you might be noticing the undefined and null. Don't worry about those. Remember that the .exec() returns error and data for values. So doing `.exec(console.log)` is the same as doing .exec(console.log(err, data))` The second method will remove the undefined message, but add null on a new line. It's up to you if you want to type more.

Exposing Sails

In sails console, type in `sails` to view a list of sails properties. You can use this to learn more about sails, override properties, or check to see if you disabled globals.

```
sails> sails
  |>   [a lifted Sails app on port 1337]
\___/  For help, see: http://links.sailsjs.org/docs

Tip: Use `sails.config` to access your app's runtime configuration.

1 Models:
User

1 Controllers:
UserController

20 Hooks:
moduleloader,logger,request,orm,views,blueprints,responses,controllers,sockets,p
ubsub,policies,services,csrf,cors,i18n,userconfig,session,grunt,http,projecthooks

sails>
```

sails debug

Attach the node debugger and lift the sails app; similar to running `node --debug app.js`. Takes the same options as `sails lift`. You can then use node-inspector to debug your app as it runs.

Example

```
$ sails debug

info: Running node-inspector on this app...
info: If you don't know what to do next, type `help`
info: Or check out the docs:
info: http://nodejs.org/api/debugger.html

info: ( to exit, type <CTRL>+<C> )

debugger listening on port 5858
```

> To use the standard (command-line) node debugger with sails, you can always just run `node debug app.js`.

sails generate

Sails ships with several *generators* to help you scaffold new projects. You can also create your own generators to handle frequent tasks, or extend functionality (for example, by creating a generator that outputs view files for your favorite templating language).

The following generators are bundled with Sails:

sails generate new <appName>

Create a new Sails project in a folder called **appName**. See `sails new` for usage options.

sails generate api <foo>

Generate **api/models/Foo.js** and **api/controllers/FooController.js**

sails generate model <foo> [attribute1:type1, attribute2:type2 ...]

Generate **api/models/Foo.js**, optionally include attributes with the specified types.

sails generate controller <foo> [action1, action2, ...]

Generate **api/controllers/FooController.js**, optionally include actions with the specified names.

sails generate adapter <foo>

Generate a **api/adapters/foo** folder containing the files necessary for building a new adapter.

sails generate generator <foo>

Generate a **foo** folder containing the files necessary for building a new generator.

sails lift

Run the Sails app in the current dir (if `node_modules/sails` exists, it will be used instead of the globally installed Sails)

Options:

- `--dev` - in development environment (the default). In the development environment Sails use *grunt-watch* to keep a eye on your files in `/assets`. If you change something (for example in one of our css-files) and reload your browser Sails will automatically show your changes. Also you views won't be cached so you can change your view-files without restarting Sails like the assets.
- `--prod` - in production environment
- `--port <portNum>` - on the port specified by `portNum` instead of the default (1337)
- `--verbose` - with verbose logging enabled
- `--silly` - with insane logging enabled

Example

```
$ sails lift

info: Starting app...

info:
info:
info:    Sails              <|
info:    v0.10.3             |\
info:                       /|.\
info:                      / || \
info:                    ,'  |'  \
info:                 .-'.-==|/_--'
info:                 `--'-------'
info:    __---___--___---___--___---___--___
info:  ____---___--___---___--___---___--___-__
info:
info: Server lifted in `/Users/mikermcneil/code/sandbox/second`
info: To see your app, visit http://localhost:1337
info: To shut down Sails, press <CTRL> + C at any time.

debug: --------------------------------------------------------
debug: :: Sat Apr 05 2014 17:03:39 GMT-0500 (CDT)

debug: Environment : development
debug: Port        : 1337
debug: --------------------------------------------------------
```

sails new

`sails new <appName>` creates a new Sails project in a folder called **appName**.

Options:

- `--no-linker` Disable automatic asset linking in your view and static HTML files (the relevant grunt tasks will not be created)
- `--no-frontend` Disable the generation of the `assets` folder and files. Views will be created with hardcopied linked resources off of sailsjs.org.
- `--template=[template language]` Use a different template language than the default (e.g. `jade`). Requires that a views generator for that language (e.g. `sails-generate-views-jade`) be installed in your global node path (e.g. `~/node_modules/` works).

> `sails new` is really just a special generator which runs `sails-generate-new`. In other words, running `sails new foo` is an alias for running `sails generate new foo`, and like any Sails generator, the actual generator module which gets run can be overridden in your global `~/.sailsrc` file.

sails version

Get the current globally installed Sails version.

Example

```
$ sails version
0.10.0-rc5
```

Request (`req`)

Sails is built on Express, and uses Node's HTTP server conventions. Because of this, you can access all of the Node and Express methods and properties on the `req` object whereever it is accessible (i.e. in your controllers, policies, and custom responses.)

A nice side effect of this compatibility is that, in many cases, you can paste existing Node.js code into a Sails app and it will work. And since Sails implements a transport-agnostic request interpreter, the code in your Sails app is WebSocket-compatible as well.

Sails adds a few methods and properties of its own to the `req` object, like `req.wantsJSON` and `req.params.all()`. These features are syntactic sugar on top of the underlying implementation, and also support both HTTP and WebSockets.

Protocol Support

The chart below describes support for the methods and properties on the Sails Request object (`req`) across multiple transports:

	HTTP	WebSockets
req.file()	:white_check_mark:	:white_large_square:
req.param()	:white_check_mark:	:white_check_mark:
req.route	:white_check_mark:	:white_check_mark:
req.cookies	:white_check_mark:	:white_large_square:
req.signedCookies	:white_check_mark:	:white_large_square:
req.get()	:white_check_mark:	:white_large_square:
req.accepts()	:white_check_mark:	:white_large_square:
req.accepted	:white_check_mark:	:white_large_square:
req.is()	:white_check_mark:	:white_large_square:
req.ip	:white_check_mark:	:white_check_mark:
req.ips	:white_check_mark:	:white_large_square:
req.path	:white_check_mark:	:white_large_square:
req.host	:white_check_mark:	:white_large_square:
req.fresh	:white_check_mark:	:white_large_square:
req.stale	:white_check_mark:	:white_large_square:

req.xhr	:white_check_mark:	:white_large_square:
req.protocol	:white_check_mark:	:white_check_mark:
req.secure	:white_check_mark:	:white_large_square:
req.session	:white_check_mark:	:white_check_mark:
req.subdomains	:white_check_mark:	:white_large_square:
req.method	:white_check_mark:	:white_check_mark:
req.originalUrl	:white_check_mark:	:white_large_square:
req.acceptedLanguages	:white_check_mark:	:white_large_square:
req.acceptedCharsets	:white_check_mark:	:white_large_square:
req.acceptsCharset()	:white_check_mark:	:white_large_square:
req.acceptsLanguage()	:white_check_mark:	:white_large_square:
req.isSocket	:white_check_mark:	:white_check_mark:
req.params.all()	:white_check_mark:	:white_check_mark:
req.socket.id	:heavy_multiplication_x:	:white_check_mark:
req.socket.join	:heavy_multiplication_x:	:white_check_mark:
req.socket.leave	:heavy_multiplication_x:	:white_check_mark:
req.socket.broadcast	:heavy_multiplication_x:	:white_check_mark:
req.transport	:white_large_square:	:white_check_mark:
req.url	:white_check_mark:	:white_check_mark:
req.wantsJSON	:white_check_mark:	:white_check_mark:

Legend

- :white_check_mark: - fully supported
- :white_large_square: - feature not yet implemented
- :heavy_multiplication_x: - unsupported due to protocol restrictions

req.accepted

Contains an array of the "media types" this request (`req`) can accept (e.g. `text/html` or `application/json`), ordered from highest to lowest quality.

Usage

```
req.accepted;
```

Example

```
req.accepted;

/*
  [ { value: 'application/json',
      quality: 1,
      type: 'application',
      subtype: 'json' },
  { value: 'text/html',
       quality: 0.5,
       type: 'text',
       subtype: 'html' } ]
*/
```

Notes

> - See the `accepts` module for the finer details of the header parsing algorithm used in Sails/Express/Koa/Connect.

req.acceptedCharsets

This property is an array that contains the acceptable charsets specified by the user agent in the request.

Usage

```
req.acceptedCharsets;
```

Details

Useful for advanced content negotiation where a client may or may not support certain character sets, such as unicode (utf-8.) This returns all of the "acceptable" charsets specified in this request's `Accept-Charset` header (see RFC-2616.)

Example

```
req.acceptedCharsets;
// -> ['utf-8', 'utf-16']
```

Notes

> - See the `accepts` module for the finer details of the header parsing algorithm used in Sails/Express/Koa/Connect.

req.acceptedLanguages

An array containing the "acceptable" response languages specified by the user agent in the "Accept-Language" header of this request (`req`).

Usage

```
req.acceptedLanguages;
```

Details

`req.acceptedLanguages` contains all the languages specified by the request's `Accept-Language` header (see RFC-2616.)

This method is used by Sails internally for its implementation of internationalization and localization. The i18n hook automatically serves different content to different locales, based on the request.

Example

```
req.acceptedLanguages;
// -> ['en-US', 'en']
```

Notes

> - See the `accepts` module for the finer details of the header parsing algorithm used in Sails/Express/Koa/Connect.
> - Browsers send the "Accept-Language" header automatically based on the user's language settings.
> - You can expect the "Accept-Language" header to exist in most requests which originate from web browsers.

req.accepts()

Checks whether this request's stated list of "accepted" media types includes the specified `type`. Returns true or false.

Usage

```
req.accepts(type);
```

Example

```
req.accepts('application/json');
// -> true
req.accepts('json');
// -> true
```

Notes

> - See the `accepts` module for the finer details of the header parsing algorithm used in Sails/Express/Koa/Connect.

req.acceptsCharset()

Returns whether this request (`req`) is able to handle a specified `characterSet` .

Usage

```
req.acceptsCharset(characterSet);
```

Details

Useful for advanced content negotiation where a client may or may not support certain character sets, such as unicode (utf-8.) This method determines whether or not a request has specified the given `characterSet` as "acceptable" its `Accept-Charset` header (see RFC-2616.)

Example

If a request is sent with a `"Accept-Charset: utf-8"` header:

```
req.acceptsCharset('utf-8');
// -> true
```

Notes

> - See the `accepts` module for the finer details of the header parsing algorithm used in Sails/Express/Koa/Connect.

req.acceptsLanguage()

Returns whether this request (`req`) considers a certain `language` "acceptable".

Usage

```
req.acceptsLanguage(language);
```

Details

`req.acceptsLanguage()` returns true if a request has specified the given `language` as "acceptable" its `Accept-Language` header (see RFC-2616.)

This method is used by Sails internally for its implementation of internationalization and localization. The i18n hook automatically serves different content to different locales, based on the request.

Example

If a request is sent with a `"Accept-Charset: utf-8"` header:

```
req.acceptsCharset('utf-8');
// -> true
```

Notes

> - See the `accepts` module for the finer details of the header parsing algorithm used in Sails/Express/Koa/Connect.
> - Browsers send the "Accept-Language" header automatically based on the user's language settings.
> - You can expect the "Accept-Language" header to exist in most requests which originate from web browsers.

req.allParams()

Returns the value of *all* parameters sent in the request, merged together into a single object. Includes parameters parsed from the url path, the query string, and the request body. See `req.param()` for details.

Usage

```
req.allParams();
```

Example

Update the product with the specified `sku`, setting new values using the parameters which were passed in:

```
var values = req.allParams();

// Don't allow `price` or `isAvailable` to be edited.
delete values.price;
delete values.isAvailable;

// At this point, `values` might look something like this:
// values ==> { displayName: 'Bubble Trouble Bubble Bath' }

Product.update({sku: sku})
.set(values)
.then(function (newProduct) {
  // ...
});
```

Notes

> - This method can also be called as `req.params.all()` - they are synonyms.

req.body

An object containing text parameters from the parsed request body, defaulting to `{}`.

By default, the request body can be url-encoded or stringified as JSON. Support for other formats, such as serialized XML, is possible using the middleware configuration.

Usage

```
req.body;
```

Notes

> - If a request contains one or more file uploads, only the text parameters sent ***before*** the first file parameter will be available in `req.body`.

req.cookies

An object containing all of the **unsigned cookies** from this request (`req`).

Usage

```
req.cookies;
```

Example

Assuming the request contained a cookie named "chocolatechip" with value "Yummy:

```
req.cookies.chocolatechip;
// "Yummy"
```

req.file()

Returns a readable Node stream of incoming multipart file uploads (an `Upstream`) from the specified `field` .

Usage

```
req.file(field);
```

Details

`req.file()` comes from Skipper, an opinionated variant of the original Connect body parser that allows you to take advantage of high-performance, streaming file uploads without any dramatic changes in your application logic.

This is a great simplification, but comes with a minor caveat: **Text parameters must be included before files in the request body.** Typically, these text parameters contain string metadata which provides additional information about the file upload.

Multipart requests to Sails should send all of their **text parameters**. before sending *any* **file parameters**. For instance, if you're building a web frontend that communicates with Sails, you should include text parameters *first* in any form upload or AJAX file upload requests. The term "text parameters" refers to the metadata parameters you might send along with the file(s) providing some additional information about this upload.

How It Works

Skipper treats *all* file uploads as streams. This allows users to upload monolithic files with minimal performance impact and no disk footprint, all the while protecting your app against nasty denial-of-service attacks involving tmp files.

When a multipart request hits your server, instead of writing temporary files to disk, Skipper buffers the request just long enough to run your app code, giving you an opportunity to "plug in" to a compatible blob receiver. If you don't "plug in" the data from a particular field, the Upstream hits its "high water mark", the buffer is flushed, and subsequent incoming bytes on that field are ignored.

Example

In a controller action or policy:

```
var SomeReceiver = require('../services/SomeReceiver');

req.file('avatar').upload( SomeReceiver(), function (err, files) {
    if (err) return res.serverError(err);
    return res.json({
      message: files.length + ' file(s) uploaded successfully!',
      files: files
    });
  });
});
```

Notes

> - Remember that the client request's **text parameters must be sent first**, before the file parameters.
> - `req.file()` supports multiple files sent over the same field, but it's important to realize that, as a consequence, the Upstream it returns is actually a stream (buffered event emitter) of potential binary streams (files).
> - If you prefer to work directly with the Upstream as a stream of streams, you can omit the `.upload()` method and bind "finish" and "error" events (or use `.pipe()`) instead. Under the covers, all `.upload()` is doing is piping the **Upstream** into the specified receiver instance, then running the specified callback when the Upstream emits either a `finish` or `error` event.

req.fresh

A flag indicating the user-agent sending this request (`req`) wants "fresh" data (as indicated by the "if-none-match", "cache-control", and/or "if-modified-since" request headers.)

If the request wants "fresh" data, usually you'll want to `.find()` fresh data from your models and send it back to the client.

Usage

```
req.fresh;
```

Example

```
if (req.fresh) {
  // The user-agent is asking for a more up-to-date version of the requested resource.
  // Let's hit the database to get some stuff and send it back.
}
```

Notes

> - See the `node-fresh` module for details specific to the implementation in Sails/Express/Koa/Connect.

req.get()

Returns the value of the specified `header` field in this request (`req`). Note that header names are case-*insensitive*.

Usage

```
req.get(header);
```

Example

Assuming `req` contains a header named 'myField' with value 'cat':

```
req.get('myField');
// -> cat
```

Notes

> - The `header` argument is case-insensitive.
> - The `header` argument treats both "referrer" and "referer" as synonyms, because sp3ll1n6.

req.headers

An object containing pre-defined/custom header given in the current request.

Usage

```
req.headers;
```

Details

Often we want to check the headers of the current request, so this can be done easily in the sails.

Example

Sample output of the `req.headers` object:

```
console.log(req.headers);

{ host: 'localhost:1337',
  connection: 'keep-alive',
  'cache-control': 'no-cache',
  'user-agent': 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) C
  accept: '*/*',
  'accept-encoding': 'gzip, deflate, sdch',
  'accept-language': 'en-US,en;q=0.8,hi;q=0.6',
  cookie: 'sdfksldddklfk; sails.sid=s%3skdlfjkj1231lsdfnsc,m' }
```

Example

And if you want to access any specific, custom or pre-defined header, it can be done with bracket notation:

```
req.headers['custom-header'];
```

or dot notation:

```
req.headers.host;
```

req.host

The hostname of this request, without the port number, as specified by its "Host" header.

Usage

```
req.host;
```

Example

If this request's "Host" header was: "ww3.staging.ibm.com:1492":

```
req.host;
// -> "ww3.staging.ibm.com"
```

req.ip

Purpose

The IP address of the client who sent this request (`req`).

> **Note:**
>
> If the `trust proxy` option is disabled in Express, this is the "remote address". Otherwise, if `trust proxy` is enabled, this is the "upstream address". See Express docs for app.set() - in Sails this can be accomplished by adding the line `sails.hooks.http.app.set('trust proxy', true);` to `config/bootstrap.js` .

Usage

```
req.ip;
```

Example

```
req.ip;
// -> "127.0.0.1"
```

req.ips

If "trust proxy" is enabled, this variable contains the IP addresses in this request's "X-Forwarded-For" header as an array of the IP address strings. Otherwise an empty array is returned.

Usage

```
req.ips;
```

Example

If a request contains a header: "X-Forwarded-For: client, proxy1, proxy2":

```
req.ips;
// -> ["client", "proxy1", "proxy2"]`

// ("proxy2" is the furthest "down-stream" IP address)
```

req.is()

Returns true if this request's declared "Content-Type" matches the specified media/mime `type`.

Specifically, this method matches the given `type` against this request's "Content-Type" header.

Usage

```
req.is(type);
```

Example

Assuming the request contains a "Content-Type" header, "text/html; charset=utf-8":

```
req.is('html');
// -> true
req.is('text/html');
// -> true
req.is('text/*');
// -> true
```

req.isSocket

A flag indicating whether or not this request (`req`) originated from a Socket.io connection.

Usage

```
req.isSocket;
```

Example

```
if (req.isSocket){
  // You're a socket.  Do cool socket stuff.
}
else {
  // Just another HTTP request.
}
```

Notes

> - Useful for allowing HTTP requests to skip calls to pubsub or WebSocket-centric methods like `subscribe()` or `watch()` that depend on an actual Socket.io request. This allows you to reuse backend code, using it for both WebSocket and HTTP clients.
> - As you might expect, `req.isSocket` doesn't need to be checked before running methods which **publish to other** connected sockets. Those methods don't depend on the request, so they work either way.

req.method

The request method (aka "verb".)

Usage

```
req.method;
```

Example

If a client sends a POST request to `/product` :

```
req.method;
// -> "POST"
```

Notes

> - All requests to a Sails server have a "method", even via WebSockets (this is thanks to the request interpreter)

req.options

`req.options` allows altering of (or providing defaults for) request parameters without modifying the original object.

Any properties provided in a custom route configuration are made available in the `req.options` object. For example, given the following in `config/routes.js`:

```
"GET /foo": {controller: 'user', action: 'myAction', owl: 'hoot'}
```

`req.options.controller`, `req.options.action` and `req.options.owl` will all be available.

Additionally, several special `req.options` objects are available for use with blueprints, specifically to programatically change the criteria and/or values that a blueprint action uses when accessing a data store. These are best used in policies to, for example, filter requested records based on the logged-in user.

Example

In a `config/policies/filterByUser.js` policy:

```
module.exports = function filterByUser (req, res, next) {

  if (req.session.user) {

    req.options.where = req.options.where || {};
    req.options.where.userId = req.session.user.id;

  }

  return next();

}
```

req.options.values

Default values for blueprint `create` and `update` actions.

> Note: Before using `req.options.value`, confirm that it exists and create it if necessary.

Example

To default to using the logged-in user's name when creating a new record:

```
// In config/policies/createWithUserName.js
module.exports = function createWithUserName (req, res, next) {

  if (req.session.user) {

    // Use existing req.options.values, or initialize it to an empty object
    req.options.values = req.options.values || {};

    // Set the default `name` for "create" and "updates" blueprints
    req.options.values.name = req.session.user.name;

  }

  return next();

}
```

Then apply this policy to the desired blueprint actions.

req.options.values.where

Default “where” criteria for user with blueprint `find` and `update` actions.

> Note: Before using `req.options.where`, confirm that it exists and create it if necessary.

Example

To default to finding only records where `userId` matches the logged-in user’s id:

```
// In config/policies/filterByUser.js
module.exports = function filterByUser (req, res, next) {

  if (req.session.user) {

    // Use existing req.options.where, or initialize it to an empty object
    req.options.where = req.options.where || {};

    // Set the default `userId` for "find" and "update" blueprints
    req.options.where.userId = req.session.user.id;

  }

  return next();

}
```

Then apply this policy to the desired blueprint actions.

req.param()

Returns the value of the parameter with the specified name.

Usage

```
req.param(name[, defaultValue]);
```

Details

`req.param()` searches the url path, query string, and body of the request for the specified parameter. If no parameter value exists anywhere in the request with the given `name`, it returns `undefined`, or the optional `defaultValue` if specified.

- url path parameters (`req.params`)
  - e.g. a request "/foo/4" to route `/foo/:id` has url path params `{ id: 4 }`
- query string parameters (`req.query`)
  - e.g. a request "/foo?email=5" has query params `{ email: 5 }`
- body parameters (`req.body`)
  - e.g. a request with a parseable body (e.g. JSON, url-encoded, or XML) has body parameters equal to its parsed value

Example

Consider a route (`POST /product/:sku`) which points to a blueprint, controller, or policy with the following code:

```
req.param('sku');
// -> 123
```

We can get the expected result by sending the `sku` parameter any of the following ways:

- `POST /product/123`
- `POST /product?sku=123`
- `POST /product`
  - with a JSON request body: `{ "sku": 123 }`

Notes

> - If you'd like to get ALL parameters from ALL sources (including the URL path, query string, and parsed request body) you can use `req.allParams()` .

req.params

An object containing parameter values parsed from the URL path.

For example if you have the route `/user/:name`, then the "name" from the URL path wil be available as `req.params.name`. This object defaults to `{}`.

Usage

```
req.params;
```

Notes

> - When a route address is defined using a regular expression, each capture group match from the regex is available as `req.params[0]`, `req.params[1]`, etc.This strategy is also applied to unnamed wild-card matches in string routes such as `/file/*`.

req.path

The URL pathname from the request URL string of the current request (`req`). Note that this is the part of the URL after and including the leading slash (e.g. `/foo/bar`), but without the query string (e.g. `?name=foo`) or fragment (e.g. `#foobar` .)

Usage

```
req.path;
```

Example

Assuming a client sends the following request:

> http://localhost:1337/donor/37?name=foo#foobar

`req.path` will be defined as follows:

```
req.path;
// -> "/donor/37"
```

req.protocol

The protocol used to send this request (`req`).

Usage

```
req.protocol;
```

Example

```
switch (req.protocol) {
  case 'http':
    // this is an HTTP request
    break;
  case 'https':
    // this is a secure HTTPS request
    break;
}
```

req.query

An object containing the parsed query-string, defaulting to `{}`.

Usage

```
req.query;
```

Example

If the request is `GET /search?q=mudslide`:

```
req.query.q
// -> "mudslide"
```

req.secure

Indicates whether or not the request was sent over a secure TLS connection (i.e. `https://` or `wss://`).

Usage

```
req.secure;
```

req.signedCookies

An object containing all the signed cookies from the request object. A signed cookie is protected against modification by the client. This protection is provided by a base64 encoded HMAC of the cookie value. When retrieving the cookie, if the HMAC signature does not match based on the cookie's value, then the cookie is not available as a member of the `req.signedCookies` object

Purpose

An object containing all of the signed cookies from this request (`req`).

Usage

```
req.signedCookies;
```

Example

Adding a signed cookie named "chocolatechip" with value "Yummy:

```
res.cookie('chocolatechip', 'Yummy', {signed:true});
```

Retrieving the cookie:

```
req.signedCookies.chocolatechip;
// "Yummy"
```

req.socket

If the current Request (`req`) originated from a connected Socket.io client, `req.socket` refers to the raw Socket.io socket instance.

Usage

```
req.socket;
```

Details

> **Warning:**
>
> `req.socket` may be deprecated in a future release of Sails. You should use the `sails.sockets.*` methods instead.

If the current request (`req`) did NOT originate from a Socket.io client, `req.socket` does not have the same meaning. In the most common scenario, HTTP requests, `req.socket` actually *does exist*, but it refers instead to the underlying TCP socket. Before using `req.socket` , you should check the `req.isSocket` flag to ensure the request arrived via a connected Socket.io client.

`req.socket.id` is a unique identifier representing the current socket. This is generated by the Socket.io server when a client first connects, and is a valid unique identifier until the socket is disconnected (e.g. if the client is a web browser, until the user closes her browser tab.)

Sails also provides direct, low-level access to all of the other methods and properties from a Socket.io `Socket` , including `req.socket` , including `req.socket.join` , `req.socket.leave` , `req.socket.broadcast` , and more. See the relevant docs in the Socket.io wiki for more information.

Example

```
if (req.isSocket) {
  // Low-level Socket.io methods and properties accessible on req.socket.
  // ...
}
else {
  // This is not a request from a Socket.io client, so req.socket
  // may or may not exist.  If this is an HTTP request, req.socket is actually
  // the underlying TCP socket.
  // ...
}
```

req.subdomains

An array of all the subdomains in this request's URL.

Usage

```
req.subdomains;
```

Example

If the requested URL was "https://ww3.staging.ibm.com":

```
req.subdomains;
// -> ['ww3', 'staging']
```

req.url

Like `req.path`, but also includes the query string suffix.

```
req.url;

// => "/search?q=worlds%20largest%20dogs"
```

Notes

> - It is worth mentioning that the URL fragment/hash (e.g. "#some/clientside/route") part of the url is not available on the server. This is an open issue with the current HTTP specification. So if you write an action to redirect from one subdomain to another, for instance, you won't be able to peek at the URL fragment in that action.
> - However, if you respond with a 302 redirect (i.e. `res.redirect()`) the user agent on the other end will preserve the URL fragment/hash and tack it on to the end of the new redirected URL. In many cases, this is exactly what you want!

req.wantsJSON

A flag indicating whether the requesting client would prefer a JSON response (as opposed to some other format, like XML or HTML.)

`req.wantsJSON` is used by all of the built-in custom responses in Sails.

Usage

```
req.wantsJSON;
```

Details

The intended purpose of `req.wantsJSON` is to provide a clean, reusable indication of whether the server should respond with JSON, or send back something else (like an HTML page or a 302 redirect.) It is not the right answer for *every* content negotiation problem, but it is a simple, go-to solution for most use cases.

For instance, for requests typed into the URL bar, all major browsers set an "Accept: text/plain;" request header. In that case, `req.wantsJSON` is false. But for many other cases, the distinction is not quite as clear. In those scenarios, Sails uses heuristics to determine the best value for `req.wantsJSON` .

Technically, `req.wantsJSON` inspects the request's `"Content-type"` , `"Accepts"` , and `"X-Requested-With"` headers to make an inference as to whether the request is expecting a JSON response. If the request did not provide enough information to know for sure, Sails errs on the side of JSON (i.e. `req.wantsJSON` will be set to `true` .)

This all makes your app more future-proof and less brittle: as best-practices for content negotiation change over time (e.g. a new type of consumer device or enterprise user-agent introduces a new header) Sails can patch `req.wantsJSON` at the framework level and modify the heuristics accordingly. Not to mention that it reduces code duplication and saves you the annoyance of manually inspecting headers in each of your routes.

Example

```
if (req.wantsJSON) {
  return res.json(data);
}
else {
  return res.view(data);
}
```

Details

Here is the specific order in which `req.wantsJSON` inspects the request. **If any of the following match, subsequent checks are ignored.**

A request "wantsJSON" if:

- if this looks like an AJAX request
- if this is a virtual request from a socket
- if this request DOESN'T explicitly want HTML
- if this request has a "json" content-type AND ALSO has its "Accept" header set
- if `req.options.wantsJSON` is truthy

Notes

> - Lower-level content negotiation is, of course, still possible using `req.is()`, `req.accepts()`, `req.xhr`, and `req.get()`.
> - As of Sails v0.10, requests originating from a WebSocket client always "want JSON".

req.xhr

A flag indicating whether the current request (`req`) appears to be an AJAX request (i.e. it was issued with its "X-Requested-With" header set to "XMLHttpRequest".)

Usage

```
req.xhr;
```

Example

```
if (req.xhr) {
  // Yup, it's AJAX alright.
}
```

Notes

> - Whenever possible, you should prefer the `req.wantsJSON` flag. Avoid writing custom content-negotiation negotiation logic into your app - it makes your code more brittle and more verbose.

Response (`res`)

Overview

Sails is built on Express, and uses Node's HTTP server conventions. Because of this, you can access all of the Node and Express methods and properties on the `res` object whereever it is accessible (i.e. in your controllers, policies, and custom responses.)

A nice side effect of this compatibility is that, in many cases, you can paste existing Node.js code into a Sails app and it will work. And since Sails implements a transport-agnostic request interpreter, the code in your Sails app is WebSocket-compatible as well.

Sails adds a few methods of its own to the `res` object, like `res.view()`. These features are syntactic sugar on top of the underlying implementation, and also support both HTTP and WebSockets.

Protocol Support

The chart below describes support for the methods and properties on the Sails Request object (`req`) across multiple transports:

The chart below describes support for the methods and properties on the Sails Response object (`res`) across multiple transports:

	HTTP	WebSockets
res.status()	:white_check_mark:	:white_check_mark:
res.set()	:white_check_mark:	:white_large_square:
res.get()	:white_check_mark:	:white_large_square:
res.cookie()	:white_check_mark:	:white_large_square:
res.clearCookie()	:white_check_mark:	:white_large_square:
res.redirect()	:white_check_mark:	:white_check_mark:
res.location()	:white_check_mark:	:white_large_square:
res.charset	:white_check_mark:	:white_check_mark:
res.send()	:white_check_mark:	:white_check_mark:
res.json()	:white_check_mark:	:white_check_mark:
res.jsonp()	:white_check_mark:	:white_check_mark:
res.type()	:white_check_mark:	:white_large_square:
res.format()	:white_check_mark:	:white_large_square:
res.attachment()	:white_check_mark:	:white_large_square:
res.sendfile()	:white_check_mark:	:white_large_square:
res.download()	:white_check_mark:	:white_large_square:
res.links()	:white_check_mark:	:white_large_square:
res.locals	:white_check_mark:	:white_check_mark:
res.render()	:white_check_mark:	:white_large_square:
res.view()	:white_check_mark:	:white_large_square:

Legend

- :white_check_mark: - fully supported
- :white_large_square: - feature not yet implemented
- :heavy_multiplication_x: - unsupported due to protocol restrictions

res.attachment()

Sets the "Content-Disposition" header of the current response to "attachment". If a `filename` is given, then the "Content-Type" will be automatically set based on the extension of the file (e.g. `.jpg` or `.html`), and the "Content-Disposition" header will be set to "filename= `filename` ".

Usage

```
res.attachment([filename]);
```

Example

```
res.attachment();
// -> response header will contain:
//   Content-Disposition: attachment

res.attachment('path/to/logo.png');
// -> response header will contain:
//   Content-Disposition: attachment; filename="logo.png"
//   Content-Type: image/png
```

res.badRequest()

This method is used to send a 400 ("Bad Request") response back down to the client indicating that the request is invalid. This usually means it contained invalid parameters or headers, or tried to do something impossible based on your app logic.

Usage

```
return res.badRequest();
```

Or:

- `return res.badRequest(data);`
- `return res.badRequest(data, pathToView);`

Details

Like the other built-in custom response modules, the behavior of this method is customizable.

By default, it works as follows:

- If the request "wants JSON" (e.g. the request originated from AJAX, WebSockets, or a REST client like cURL), Sails will send the provided error `data` as JSON. If no `data` is provided a default response body will be sent (the string `"Bad Request"`).
- If the request *does not* "want JSON" (e.g. a URL typed into a web browser), Sails will attempt to serve one of your views.
  - If a specific `pathToView` was provided, Sails will attempt to use that view.
  - Alternatively if `pathToView` was *not* provided, Sails will try to guess an appropriate view (see `res.view()` for details). If Sails cannot guess a workable view, it will just send JSON.
  - If Sails serves a view, the `data` argument will be accessible as a view local: `data`.

Example

Using the default view:

```
if ( req.param('amount') < 500 )
  return res.badRequest(
    'Transaction limit exceeded. Please try again with an amount less than $500.'
  );
}
```

With a custom view:

```
if ( req.param('amount') < 500 )
  return res.badRequest(
    'Transaction limit exceeded. Please try again with an amount less than $500.',
    'salesforce/leads/edit'
  );
}
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).
> - `res.badRequest()` (like other userland response methods) can be overridden or modified. It runs the response method defined in `/responses/badRequest.js`, which is bundled automatically in newly generated Sails apps. If a `badRequest.js` response method does not exist in your app, Sails will implicitly use the default behavior.
> - This method is called automatically if a call to `req.validate()` fails any of its validation checks.
> - By default, the specified error (`err`) will be excluded if the app is running in the "production" environment (i.e. `process.env.NODE_ENV === 'production'`).

res.clearCookie()

Clears cookie (`name`) in the response.

Usage

```
res.clearCookie(name [,options]);
```

Details

The path option defaults to "/".

Example

```
res.cookie('name', 'tobi', { path: '/admin' });
res.clearCookie('name', { path: '/admin' });
```

res.cookie()

Sets a cookie with name (`name`) and value (`value`) to be sent along with the response.

Usage

```
res.cookie(name, value [,options]);
```

Details

The "path" option defaults to "/".

The "maxAge" option is a convenience option for setting "expires" relative to the current time in milliseconds. The following is equivalent to the previous example.

```
res.cookie('rememberme', '1', { maxAge: 900000, httpOnly: true })
```

An object may be passed which is then serialized as JSON, which is automatically parsed by the bodyParser() middleware.

```
res.cookie('cart', { items: [1,2,3] });
res.cookie('cart', { items: [1,2,3] }, { maxAge: 900000 });
```

Signed cookies are also supported through this method. Simply pass the signed option. When given res.cookie() will use the secret passed to express.cookieParser(secret) to sign the value.

```
res.cookie('name', 'tobi', { signed: true });
```

Example

```
res.cookie('name', 'tobi', {
  domain: '.example.com',
  path: '/admin',
  secure: true
});

res.cookie('rememberme', '1', {
  expires: new Date(Date.now() + 900000),
  httpOnly: true
});
```

res.forbidden()

This method is used to send a 403 ("Forbidden") response back down to the client indicating that the request is not allowed. This usually means the user-agent tried to do something it was not allowed to do, like change the password of another user.

Usage

```
return res.forbidden();
```

Or:

- `return res.forbidden(data);`
- `return res.forbidden(data, pathToView);`

Details

Like the other built-in custom response modules, the behavior of this method is customizable.

By default, it works as follows:

- If the request "wants JSON" (e.g. the request originated from AJAX, WebSockets, or a REST client like cURL), Sails will send the provided error `data` as JSON. If no `data` is provided a default response body will be sent (the string `"Forbidden"`).
- If the request *does not* "want JSON" (e.g. a URL typed into a web browser), Sails will attempt to serve one of your views.
  - If a specific `pathToView` was provided, Sails will attempt to use that view.
  - Alternatively if `pathToView` was *not* provided, Sails will serve a default error page (the view located at `views/403.ejs`). If that view does not exist, Sails will just send JSON.
  - If Sails serves a view, the `data` argument will be accessible as a view local: `data`.

Example

Using the default view:

```
if ( !req.session.canEditSalesforceLeads ) {
  return res.forbidden('Write access required');
}
```

With a custom view:

```
if ( !req.session.canEditSalesforceLeads ) {
  return res.forbidden(
    ''Write access required'',
    'salesforce/leads/edit'
  );
}
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).
> - `res.forbidden()` (like other userland response methods) can be overridden or modified. It runs the response method defined in `/responses/forbidden.js`, which is bundled automatically in newly generated Sails apps. If a `forbidden.js` response method does not exist in your app, Sails will implicitly use the default behavior.
> - If `pathToView` refers to a missing view, this method will respond as if the request "wants JSON". +By default, the specified error (`err`) will be excluded if the app is running in the "production" environment (i.e. `process.env.NODE_ENV === 'production'`).

res.get()

Returns the current value of the specified response header (`header`).

Usage

```
res.get(header);
```

Example

```
res.get('Content-Type');
// -> "text/plain"
```

Notes

> - The `header` argument is case-insensitive. +Response headers can be changed up until the response is sent - see `res.set()` .

res.json()

Sends a JSON response composed of a stringified version of the specified `data` .

Usage

```
return res.json([statusCode, ] data);
```

Details

This method is identical to res.send() when an object or array is passed, however it may be used for explicit JSON conversion of non-objects (null, undefined, etc), though these are technically not valid JSON.

Example

```
res.json(null)
res.json({ user: 'tobi' })
res.json(500, { error: 'message' })
```

Notes

> - Don't forget this method's name is all lowercase.
> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).

res.jsonp()

Send a JSON or JSONP response.

Identical to `res.json()`, except if a "callback" parameter exists, a JSONP response will be sent instead, using the value of the "callback" parameter as the name of the function wrapper.

Usage

```
return res.jsonp([statusCode, ] data);
```

Example

```
return res.jsonp({
  users: [{
    name: 'Thelma',
    id: 1
  }, {
    name: 'Leonardo'
    id: 2
  }]
});
```

Notes

> - Don't forget this method's name is all lowercase.
> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).

res.location()

Sets the "Location" response header to the specified URL expression (`url`).

Usage

res.location(url);

Example

```
res.location('/foo/bar');
res.location('foo/bar');
res.location('http://example.com');
res.location('../login');
res.location('back');
```

Notes

> - You can use the same kind of URL expressions as in res.redirect().

res.negotiate()

Given an error (`err`), send an appropriate error response back down to the client. Especially handy for handling potential validation errors from Model.create() or Model.update().

Usage

```
return res.negotiate(err);
```

Details

Like the other built-in custom response modules, the behavior of this method is customizable.

`res.negotiate()` examines the provided error (`err`) and determines the appropriate error-handling behavior from one of the following methods:

- `res.badRequest()` (400)
- `res.forbidden()` (403)
- `res.notFound()` (404)
- `res.serverError()` (500)

The determination is made based on `err` 's "status" property. If a more specific diagnosis cannot be determined (e.g. `err` doesn't have a "status" property, or it's a string), Sails will default to `res.serverError()` .

Example

```
// Add Fido's birthday to the database:
Pet.update({name: 'fido'})
  .set({birthday: new Date('01/01/2010')})
  .exec(function (err, fido) {
    if (err) return res.negotiate(err);
    return res.ok(fido);
   });
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).
> - `res.negotiate()` (like other userland response methods) can be overridden - just define a response module (`/responses/negotiate.js`) and export a function definition.
> - This method is used as the default handler for uncaught errors in Sails. That means it is called automatically if an error is thrown in *any* request handling code, *but only within the initial step of the event loop*. You should always specifically handle errors that might arise in callbacks/promises from asynchronous code.

res.notFound()

Sends a 404 ("Not Found") response using either res.json() or res.view(). Called automatically when Sails receives a request which doesn't match any of its explicit routes or route blueprints (i.e. serves the 404 page).

When called manually from your app code, this method is normally used to indicate that the user-agent tried to find, update, or delete something that doesn't exist.

Usage

```
return res.notFound();
```

Or:

- `return res.notFound(data);`
- `return res.notFound(data, pathToView);`

Details

Like the other built-in custom response modules, the behavior of this method is customizable.

By default, it works as follows:

- If the request "wants JSON" (e.g. the request originated from AJAX, WebSockets, or a REST client like cURL), Sails will send the provided error `data` as JSON. If no `data` is provided a default response body will be sent (the string `"Not Found"`).
- If the request *does not* "want JSON" (e.g. a URL typed into a web browser), Sails will attempt to serve one of your views.
  - If a specific `pathToView` was provided, Sails will attempt to use that view.
  - Alternatively if `pathToView` was *not* provided, Sails will try to guess an appropriate view (see `res.view()` for details). If Sails cannot guess a workable view, it will just send JSON.
  - If Sails serves a view, the `data` argument will be accessible as a view local: `data` .

Example

Using the default view:

```
return res.notFound();
```

With a custom view:

```
Pet.findOne()
.where(name: 'fido')
.exec(function(err, fido) {
  if (err) return res.serverError(err);
  if (!fido) return res.notFound(undefined,'pet/sorry-that-pet-has-moved');
  // ...
})
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).
> - `res.notFound()` (like other userland response methods) can be overridden or modified. It runs the response method defined in `/responses/notFound.js`, which is bundled automatically in newly generated Sails apps. If a `notFound.js` response method does not exist in your app, Sails will implicitly use the default behavior.
> - If `pathToView` refers to a missing view, this method will respond as if the request "wants JSON". +By default, the specified error (`err`) will be excluded if the app is running in the "production" environment (i.e. `process.env.NODE_ENV === 'production'`).

res.ok()

Send a 200 ("OK") response back down to the client with the provided data. Performs content-negotiation on the request and calls either `res.json()` or `res.view()`.

Usage

```
return res.ok();
```

Or:

- `return res.ok(data);`
- `return res.ok(data, pathToView);`

Details

Like the other built-in custom response modules, the behavior of this method is customizable.

By default, it works as follows:

- If the request "wants JSON" (e.g. the request originated from AJAX, WebSockets, or a REST client like cURL), Sails will send the provided `data` as JSON. If no `data` is provided a default response body will be sent (the string `"OK"`).
- If the request *does not* "want JSON" (e.g. a URL typed into a web browser), Sails will attempt to serve one of your views.
    - If a specific `pathToView` was provided, Sails will attempt to use that view.
    - Alternatively if `pathToView` was *not* provided, Sails will try to guess an appropriate view (see `res.view()` for details). If Sails cannot guess a workable view, it will fall back and send JSON.
    - If Sails serves a view, the `data` argument will be accessible as a view local: `data`.

Example

```
return res.ok({
  name: 'Loïc',
  occupation: 'developer'
});
```

If the request originated from a socket or AJAX request, the response sent from the usage above would contain the following JSON:

```
{
  "name": "Loïc",
  "occupation": "developer"
}
```

Alternatively, if the code that calls `res.ok()` was located somewhere where a view file could be guessed, that view would be served, with with Loïc available as the `data` local. For example if `res.ok()` was called in `UserController.update`, then we might create the following view as `views/user/update.ejs`:

```
<input type="text" placeholder="Name" value="<%= data.name %>"/>
<input type="text" placeholder="Occupation" value="<%= data.occupation %>"/>
```

If the code that calls `res.ok()` is not in a controller action, a conventional view cannot be guessed, so Sails will just send back JSON instead.

Finally, if a custom `pathToView` is provided as the second argument, Sails will always use that view instead of guessing, e.g. the following usage will compile and respond with a view located in `views/user/detail.ejs`:

```
return res.ok({
  name: 'Loïc',
  occupation: 'developer'
}, 'user/detail');
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).
> - `res.ok()` (like other userland response methods) can be overridden or modified. It runs the response method defined in `api/responses/ok.js`, which is bundled automatically in newly generated Sails apps. If an `ok.js` response method does not exist in your app, Sails will implicitly use the default behavior.
> - This method is used by blueprint actions to send a success response. Therefore as you might expect, it is a great place to marshal response data for clients which expect data in a specific format, i.e. to convert data to XML, or it wrap in an additional object (for Ember clients).

res.redirect()

Redirect the requesting user-agent to the given absolute or relative url.

Usage

```
return res.redirect(url);
```

Arguments

	Argument	Type	Details
1	`url`	((string))	A URL expression (see below for complete specification). e.g. `"http://google.com"` or `"/login"`

Details

Sails/Express/Koa/Connect support a few forms of redirection, first being a fully qualified URI for redirecting to a different domain:

```
return res.redirect('http://google.com');
```

The second form is the domain-relative redirect. For example, if you were on http://example.com/admin/post/new, the following redirect to `/admin` would land you at http://example.com/admin:

```
return res.redirect('/checkout');
```

Pathname relative redirects are also possible. If you were on http://example.com/admin/post/new, the following redirect would land you at http//example.com/admin/post:

```
return res.redirect('..');
```

The final special-case is a back redirect, which allows you to redirect a request back where it came from using the "Referer" (or "Referrer") header (if omitted, redirects to `/` by default)

```
return res.redirect('back');
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).
> - When your app calls `res.redirect()`, Sails sends a response with status code 302. This instructs the user-agent to send a new request to the indicated URL. There is no way to *force* a user-agent to follow redirects, but most clients play nicely.
> - In general, you should not need to use `res.redirect()` if a request "wants JSON" (i.e. `req.wantsJSON`).
> - If a request originated from a Socket.io client, it always "wants JSON". If you do call `res.redirect(http://sailsjs.org/documentation/reference/res/res.redirect.html)` for a socket request, Sails reroutes the request internally on the server, effectively "forcing" the redirect to take place (i.e. instead of sending a 302 status code, the server simply creates a new request to the redirect URL).
>   - As a result, redirects to external domains are not supported for socket requests (although this is technically possible by proxying).
>   - This behavior may change to more closely reflect HTTP in future versions of Sails.

res.send()

Send a simple response. `statusCode` defaults to 200 ("OK").

This method is used in the underlying implementation of most of the other terminal response methods.

Usage

```
return res.send([statusCode,] body);
```

Details

This method performs a myriad of useful tasks for simple non-streaming responses such as automatically assigning the Content-Length unless previously defined and providing automatic HEAD and HTTP cache freshness support.

When a Buffer is given the Content-Type is set to "application/octet-stream" unless previously defined as shown below:

```
res.set('Content-Type', 'text/html');
res.send(new Buffer('some html'));
```

When a String is given the Content-Type is set to "text/html":

```
res.send('some html');
```

When an Array or Object is given Express will respond with the JSON representation:

```
res.send({ user: 'tobi' })
res.send([1,2,3])
```

Finally when a Number is given without any of the previously mentioned bodies, then a response body string is assigned for you. For example 200 will respond will the text "OK", and 404 "Not Found" and so on.

```
res.send(200)
res.send(404)
res.send(500)
```

Example

```
res.send(new Buffer('whoop'));
res.send({ some: 'json' });
res.send('some html');
res.send(404, 'Sorry, we cannot find that!');
res.send(500, { error: 'something blew up' });
res.send(200);
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).

res.serverError()

This method is used to send a 500 ("Server Error") response back down to the client indicating that some kind of server error occurred.

Usage

```
return res.serverError();
```

Or:

- `return res.serverError(data);`
- `return res.serverError(data, pathToView);`

Details

Like the other built-in custom response modules, the behavior of this method is customizable.

By default, it works as follows:

- If the request "wants JSON" (e.g. the request originated from AJAX, WebSockets, or a REST client like cURL), Sails will send the provided error `data` as JSON. If no `data` is provided a default response body will be sent (the string `"Server Error"`).
- If the request *does not* "want JSON" (e.g. a URL typed into a web browser), Sails will attempt to serve one of your views.
    - If a specific `pathToView` was provided, Sails will attempt to use that view.
    - Alternatively if `pathToView` was *not* provided, Sails will serve a default error page (the view located at `views/500.ejs`). If that view does not exist, Sails will just send JSON.
    - If Sails serves a view, the `data` argument will be accessible as a view local: `data`.

Example

Using the default error view:

```
return res.serverError('Salesforce could not be reached');
```

With a custom view:

```
return res.serverError(
  'Salesforce could not be reached',
  'salesforce/leads/edit'
);
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).
> - `res.serverError()` (like other userland response methods) can be overridden or modified. It runs the response method defined in `/responses/serverError.js`, which is bundled automatically in newly generated Sails apps. If a `serverError.js` response method does not exist in your app, Sails will implicitly use the default behavior.
> - If `pathToView` refers to a missing view, this method will respond as if the request "wants JSON". +By default, the specified error (`err`) will be excluded if the app is running in the "production" environment (i.e. `process.env.NODE_ENV === 'production'`).

res.set()

Sets specified response header (`header`) to the specified value (`value`).

Alternatively, you can pass in a single object argument (`headers`) to set multiple header fields at once, where the keys are the header field names, and the corresponding values are the desired values.

Usage

```
res.set(header, value);
```

-or-

```
res.set(headers);
```

Example

```
res.set('Content-Type', 'text/plain');

res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
  'ETag': '12345'
})
```

res.status()

Set the status code of this response.

Usage

```
res.status(200);
```

Example

```
res.status(404);
res.send('oops');
```

Notes

> - The status code may be set up until the response is sent.
> - `res.status()` is effectively just a chainable alias of node's ' `res.statusCode=` .

res.type()

Sets the "Content-Type" response header to the specified `type`.

This method is pretty forgiving (see examples below), but note that if `type` contains a `"/"`, `res.type()` assumes it is a MIME type and interprets it literally.

Usage

```
res.type(type);
```

Example

```
res.type('.html');
res.type('html');
res.type('json');
res.type('application/json');
res.type('png');
```

res.view()

Respond with an HTML page.

Usage

```
return res.view(pathToView, locals);
```

Or:

- `return res.view(pathToView);`
- `return res.view(locals);`
- `return res.view();`

Uses the configured view engine to compile the view template at `pathToView` into HTML. If `pathToView` is not provided, serves the conventional view based on the current controller and action.

The specified `locals` are merged with your configured app-wide locals, as well as certain built-in locals from Sails and/or your view engine, then passed to the view engine as data.

Arguments

	Argument	Type	Details
1	`pathToView`	((string))	The path to the desired view file relative to your app's `views` folder (usually `views/`), without the file extension (e.g. `.ejs`), and with no trailing slash. Defaults to "identityOfController/nameOfAction".
2	`locals`	((object))	Data to pass to the view template. These explicitly specified locals will be merged in to Sails' built-in locals and your configured app-wide locals. Defaults to `{}`.

Example

Consider a conventionally configured Sails app with a call to `res.view()` in the `cook()` action of its `OvenController.js`.

With no `pathToView` argument, `res.view()` will decide the path by combining the identity of the controller (`oven`) and the name of the action (`cook`):

```
return res.view();
// -> responds with `views/oven/cook.ejs`
```

Here's how you would load the same view using an explicit `pathToView` :

```
return res.view('oven/cook');
// -> responds with `views/oven/cook.ejs`
```

Finally, here's a more involved example demonstrating how `res.view` can be combined with Waterline queries:

```
// Find the 5 hottest oven brands on the market
Oven.find().sort('heat ASC').exec(function (err, ovens){
  if (err) return res.serverError(err);

  return res.view('oven/top5', {
    hottestOvens: ovens
  });
  // -> responds using the view at `views/oven/top5.ejs`,
  // and with the oven data we looked up as view locals.
  //
  // e.g. in the view, we might have something like:
  // ...
  // <% _.each(hottestOvens, function (aHotOven) { %>
  //  <li><%= aHotOven.name %></li>
  // <% }) %>
  // ...
});
```

Notes

> - This method is **terminal**, meaning it is generally the last line of code your app should run for a given request (hence the advisory usage of `return` throughout these docs).
> - `res.view()` reads a view file from disk, compiles it into HTML, then streams it back to the client. If you already have the view in memory, or don't want to stream the compiled HTML directly back to the client, use `sails.hooks.views.render()` instead.
> - `res.view()` always looks for the *lowercased* version of a view filename. For example, if your controller is `FooBarController` and your action is `Baz`, `res.view()` will attempt to find `views/foobar/baz.ejs`. On *case-sensitive* filesystems (e.g. Ubuntu Linux), this can lead to unexpected errors locating views if they are saved with capital letters. For this reason, it is recommended that you always save your views and view folders in lowercase.

Configuration (`sails.config`)

The `sails.config` object contains the runtime values of your app's configuration. It is assembled automatically when Sails loads your app; merging together command-line arguments, environment variables, your `.sailsrc` file, and the configuration objects exported from any and all modules in your app's `config/` directory.

More specifically, when you load your app, whether that's using `node app`, programmatic usage inside of a script, or `sails lift`, Sails will look in a few different places for configuration. Here they are listed in order of descending priority:

- an optional object of configuration overrides passed-in programmatically
- a local `.sailsrc` file in your app's directory, or the first found looking in `../`, `../../` etc.
- a global `.sailsrc` file in your home folder (e.g. `~/.sailsrc`)
- command-line arguments (parsed by minimist)
- environment variables (prefixed with `sails_`, using double underlines to indicate dots: e.g. `sails_port=1492`, `sails_models__connection=somePostgresqlServer`, and/or `sails_connections__somePostgresqlServer__password=l0lguyz`)
- files in your app's `config/` directory (if one exists), with `config/local.js` taking priority. Remember that, other than `local.js` (which takes priority), the file names are just for convention: the configuration you export from each file gets deep-merged together with everything else into one big dictionary (`sails.config`).

> ## The recommended solution for production
>
> Environment variables are one of the most powerful ways to configure your Sails app. Since you can customize just about any setting (as long as it's JSON-serializable), this approach solves a number of problems, and is our core team's recommended strategy for production deployments. Here are a few:
>
> - Using environment variables means you don't have to worry about checking in your production database credentials, API tokens, etc.
> - This makes changing Postgresql hosts, Mailgun accounts, S3 credentials, and other maintenance straightforward, fast, and easy; plus you don't need to change any code or worry about merging in downstream commits from other people on your team
> - Depending on your hosting situation, you may be able to manage your production configuration through a UI (most PaaS providers like Heroku or Modulus support this, as does Azure Cloud.)

Miscellaneous (`sails.config.*`)

For a conceptual overview of configuration in Sails, see http://sailsjs.org/documentation/concepts/Configuration.

This page is a quick reference of assorted configuration topics that don't fit elsewhere, namely top-level properties on the sails.config object. Many of these properties are best set on a per-environment basis, or in your config/local.js. To set them globally for your app, create a new file in the `config` folder (e.g. `config/misc.js`) and add them there.

`sails.config.port`

The `port` setting determines which TCP port your app will be deployed on. Ports are a transport-layer concept designed to allow many different networking applications to run at the same time on a single computer.

By default, if it’s set, Sails uses the `PORT` environment variable. Otherwise it falls back to port 1337. In production, you’ll probably want to change this setting to 80 (http://) or 443 (https://) if you have an SSL certificate.

More about ports: http://en.wikipedia.org/wiki/Port_(computer_networking)

`sails.config.explicitHost`

By default, Sails will assume `localhost` as the host that will be listening for incoming requests. This will work in the majority of hosting environments you encounter, but in some cases (OpenShift being one example) you'll need to explicitly declare the host name of your Sails app. Setting `explicitHost` tells Sails to listen for requests on that host instead of `localhost` .

`sails.config.proxyHost` and `sails.config.proxyPort`

If your site will ultimately be served by a proxy, you may want to set `proxyHost` to ensure that calls to `sails.getBaseurl()` return the expected host. For example, if you deploy a Sails app on Modulus.io, the ultimate URL for your site will be something like `http://mysite-12345.onmodulus.net` . If you were to use `sails.getBaseurl()` to construct a URL in your app code, however, it would return something like `http://localhost:8080` . Using `proxyHost` and `proxyPort` allow you to specify the host name and port of the proxy server that will be serving your app. This ensure that any links created using `sails.getBaseurl()` are correct.

`sails.config.environment`

> **Important**
>
> The NODE_ENV environment variable is usually a better idea than setting `sails.config.environment` manually, since it's a generic Node convention. The `sails.config.environment` setting may be deprecated in Sails v1.0.

The runtime “environment” of your Sails app is either ‘development’ or ‘production’.

In development, your Sails app will go out of its way to help you (for instance you will receive more descriptive error and debugging output).

In production, Sails configures itself (and its dependencies) to optimize performance. You should always put your app in production mode before you deploy it to a server -- this helps ensure that your Sails app remains stable, performant, and scalable.

By default, Sails sets its environment using the `NODE_ENV` environment variable. If `NODE_ENV` is not set, Sails will run in the ‘development’ environment.

`sails.config.hookTimeout`

Set a global timeout for Sails hooks, in milliseconds. Sails will give up trying to lift if any hook takes longer than this to load. Defaults to `20000`.

`sails.config.keepResponseErrors`

By default, convenience functions `badRequest`, `forbidden`, `notFound`, and `serverError` will clear the response body when the environment is "production". This behavior may be undesirable in certain cases, such as exposing underlying Waterline validation errors to clients while responding through `badRequest`.

Set `keepResponseErrors` to `true` to ensure Sails preserves the response body for these functions.

sails.config.blueprints

By default, Sails controllers automatically bind routes for each of their functions. Additionally, each controller will automatically bind routes for a CRUD API controlling the model which matches its name, if one exists.

Properties

Property	Type	Default	Details
`actions`	((boolean))	true	Whether routes are automatically generated for every action in your controllers (also maps `index` to `/:controller`) '/:controller', '/:controller/index', and '/:controller/:action'
`rest`	((boolean))	true	Automatic REST blueprints enabled? e.g. `'get /:controller/:id?'` `'post /:controller'` `'put /:controller/:id'` `'delete /:controller/:id'`
`shortcuts`	((boolean))	true	These CRUD shortcuts exist for your convenience during development, but you'll want to disable them in production.: `'/:controller/find/:id?'` , `'/:controller/create'` , `'/:controller/update/:id'` , and `'/:controller/destroy/:id'`
`prefix`	((string))	''	Optional mount path prefix for blueprints (the automatically bound routes in your controllers) e.g. '/api/v2'
`restPrefix`	((string))	''	Optional mount path prefix for RESTful blueprints (the automatically bound RESTful routes for your controllers and models) e.g. '/api/v2'. Will be joined to your `prefix` config. e.g. `prefix: '/api'` and `restPrefix: '/rest'` , RESTful actions will be available under `/api/rest`
`pluralize`	((boolean))	false	Optionally use plural controller names in blueprint routes, e.g. `/users` for `api/controllers/UserController.js` .
`populate`	((boolean))	true	Whether the blueprint controllers should populate model fetches with data from other models which are linked by associations. If you have a lot of data in one-to-many associations, leaving this on may result in

			very heavy api calls.
`defaultLimit`	((integer))	30	The default number of records to show in the response from a "find" action. Doubles as the default size of populated arrays if `populate` is `true` .
`autoWatch`	((boolean))	true	Whether to run `Model.watch()` in the `find` and `findOne` blueprint actions. Can be overridden on a per-model basis.
`jsonp`	((boolean))	false	Optionally wrap blueprint JSON responses in a JSONP callback using `res.jsonp()` from Express 3.

sails.config.bootstrap

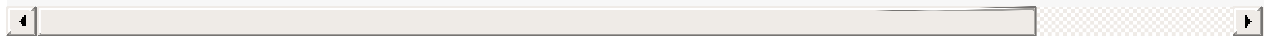
What is this?

This is an asynchronous boostrap function that runs before your Sails app gets lifted (i.e. starts up). This gives you an opportunity to set up your data model, run jobs, or perform some special logic.

Description

```
module.exports.bootstrap = function (cb) {

  // It's very important to trigger this callback method when you are finished
  // with the bootstrap!  (otherwise your server will never lift, since it's waiting on t
  cb();
};
```



sails.config.connections

What is this?

Adapters are the middle man between your Sails app and some kind of storage (typically a database)

Global connections are configured in the `connections.js` file located in your project's `config` directory. You can also specify connections in your `config/local.js` or environment-specific config files.

Sails adapters have been written for a variety of popular databases such as MySQL, Postgres and Mongo. You can find a list of supported adapters here.

Example

To use the `sails-memory` adapter (useful for DEVELOPMENT ONLY), first install the module with `npm install sails-memory`, then define it in `connections.js`:

Here is an example adapter configuration file

`myApp/config/connections.js`

```
module.exports.connections = {
  // sails-disk is installed by default.
  localDiskDb: {
    adapter: 'sails-disk'
  },
  memory: {
    adapter: 'sails-memory'
  }
};
```

If you wanted to set `memory` as the default adapter for your models, you would do this.
`myApp/config/models.js`

```
module.exports.models = {

  connection: 'memory'
};
```

> Keep in mind that options you define directly in your model definitions will override these settings. Prior to v0.10, adapters were defined in `myApp/config/Adapters.js`. See v0.9 docs for more info.

Multiple connections for an adapter

You can set up more than one connection using the same adapter. For example, if you had two mysql databases, you could configure them as:

```
module.exports.connections = {
  localMysql: {
    adapter: 'sails-mysql',
    user: 'root',
    host: 'localhost',
    database: 'someDbase'
  },
  remoteMysql: {
    adapter: 'sails-mysql',
    user: 'remoteUser',
    password: 'remotePassword',
    host: 'http://remote-mysql-host.com',
    database: 'remoteDbase'
  }
};
```

> **Note** If *any* connection to an adapter is used by a model, then *all* connections to that adapter will be loaded on `sails.lift`, whether or not models are actually using them. In the example above, if a model was configured to use the `localMysql` connection, then both `localMysql` and `remoteMysql` would attempt to connect at run time. It is therefore good practice to split your connection configurations up by environment and save them to the appropriate environment-specific config files, or else comment out any connections that you don't want active.

sails.config.cors

Configuration for Sails' built-in support for Cross-Origin Resource Sharing. CORS specifies how HTTP requests to your app originating from foreign domains should be treated. It is primarily used to allow third-party sites to make AJAX requests to your app, which are normally blocked by browsers following the same-origin policy.

These options are conventionally set in the **config/cors.js** configuration file. Note that these settings (with the exception of `allRoutes`) can be changed on a per-route basis in the **config/routes.js** file.

Properties

Property	Type	Default	Details
`allRoutes`	((boolean))	false	Indicates whether the other CORS configuration settings should apply to every route in the app by default.
`origin`	((string))	`*`	Comma-delimited list of default hosts (beginning with http:// or https://) to give access to, or `*` to allow all domains CORS access. If `allRoutes` is `true` and `origin` is `*`, then your app will be fully accessible to sites hosted on foreign domains (except for routes which have their own CORS settings).
`methods`	((string))	GET, POST, PUT, DELETE, OPTIONS, HEAD	Comma-delimited list of methods that are allowed to be used in CORS requests. This is only used in response to preflight requests, so the inclusion of GET, POST, OPTIONS and HEAD, although customary, is not necessary.
`headers`	((string))	content-type	Comma-delimited list of headers that are allowed to be sent with CORS requests. This is only used in response to preflight requests.
`exposeHeaders`	((string))	`''`	List of headers that browsers will be allowed to access. See access-control-expose-headers.
`credentials`	((boolean))	true	Indicates whether cookies can be shared in CORS requests.
`securityLevel`	((integer))	0	Indicates how Sails should respond to requests from disallowed origins. In normal mode (0), Sails processes all requests normally, simply setting the appropriate CORS headers and leaving it to the client to determine how to handle the response. In high mode (1), Sails will send back a 403 response to requests from disallowed origins, if the origin starts with http or https. In very high mode (2), Sails will send back a 403 response to requests from disallowed origins, regardless of the origin protocol. See Security Levels in the CORS concepts documentation for more info.

Custom route config example

The following will allow cross-origin AJAX GET, PUT and POST requests to `/foo/bar` from sites hosted `http://foobar.com` and `https://owlhoot.com`. DELETE requests, or requests from sites on any other domains, will be blocked by the browser.

```
'/foo/bar': {
  target: 'FooController.bar',
  cors: {
    origin: 'http://foobar.com,https://owlhoot.com',
    methods: 'GET,PUT,POST,OPTIONS,HEAD'
  }
}
```

sails.config.csrf

Configuration for Sails' built-in CSRF protection middleware. These options are conventionally set in the `config/csrf.js` configuration file. See the docs on Cross-Site Request Forgery in the security section for detailed usage instructions.

This option protects your Sails app against cross-site request forgery (or CSRF) attacks. A would-be attacker needs not only a user's session cookie, but also this timestamped, secret CSRF token, which is refreshed/granted when the user visits a URL on your app's domain.

This allows you to have certainty that your users' requests haven't been hijacked, and that the requests they're making are intentional and legitimate.

Properties

Property	Type	Default	Details
`csrf`	((boolean)) or ((object))	false	CSRF protection is disabled by default to facilitate development. To turn it on, just set `sails.config.csrf` to true, or to an object as described below.

csrf object settings

Besides `true` and `false` , you can set `sails.config.csrf` to an object with the following properties:

Property	Type	Default	Details
`grantTokenViaAjax`	((boolean))	true	Whether to activate the **/csrfToken** route, which will return the current CSRF token value which can then be used in AJAX requests.
`origin`	((string))	''	Comma-delimited list of origins that are allowed to access the CSRF token via the **/csrfToken** route. This is separate from the other CORS settings, which *do not apply* to **/csrfToken**.
`routesDisabled`	((string))	''	Comma-delimited list of routes where CSRF protection is disabled.

sails.config.globals

Configuration for the global variables that Sails exposes to its Node process. The options are conventionally specified in the `config/globals.js` configuration file.

Properties

Property	Type	Default	Details
`sails`	((boolean))	`true`	Expose the `sails` instance representing your app. If this is disabled, you can still get access via `req._sails`.
`models`	((boolean))	`true`	Expose each of your app's models as global variables (using their "globalId"). E.g. a model defined in `api/models/User.js` would have a globalId of `User` by default. If this is disabled, you can still access your models via `sails.models.*`.
`services`	((boolean))	`true`	Expose each of your app's services as global variables (using their "globalId"). E.g. a service defined in `api/services/NaturalLanguage.js` would have a globalId of `NaturalLanguage` by default. If this is disabled, you can still access your services via `sails.services.*`.
`_`	((boolean))	`true`	Expose the `lodash` installed in Sails core as a global variable. If this is disabled, like any other node module you can always run `npm install lodash --save`, then `var _ = require('lodash')` at the top of any file.
`async`	((boolean))	`true`	Expose the `async` installed in Sails core as a global variable. If this is disabled, like any other node module you can always run `npm install async --save`, then `var async = require('async')` at the top of any file.

Notes

> - To disable all global variables, you can set `sails.config.globals` to `false`.

sails.config.http

Configuration for your app's underlying HTTP server. These properties are conventionally specified in the `config/http.js` configuration file.

Properties

Property	Type	Default	Details
`middleware`	((object))	See conventional defaults for HTTP middleware	A configuration object of all HTTP middleware functions your app will run on every incoming HTTP request. All Express or Connect middleware is supported. Example
`middleware.order`	((array))	See conventional defaults for HTTP middleware order	The order in which middleware should be run for HTTP request (the Sails router, which runs the appropriate explicit routes, policies, controllers, etc. from your app is invoked by the "router" middleware).
`cache`	((number))	`cache: 31557600000`	The number of milliseconds to cache flat files on disk being served by Express static middleware (by default, these files are in `.tmp/public`) The HTTP static cache is only active in a 'production' environment (default 1 year), since that's the only time Express will cache flat-files.
`serverOptions`	((object))	TODO	TODO

Notes

> - Note that this HTTP middleware stack configured in `sails.config.http.middleware` is only applied to true HTTP requests-- it is ignored when handling virtual requests (e.g. sockets)
> - You cannot define a custom middleware function with the key `order` (since `sails.config.http.middleware.order` has special meaning)

sails.config.i18n

Configuration for Sails' built-in internationalization & localization features. For more information see the concepts section on internationalization.

Properties

Property	Type	Default	Details
`locales`	((array))	['en','es','fr','de']	List of supported locale codes
`localesDirectory`	((string))	'/config/locales'	The project-relative path to the folder containing your locale translations (i.e. stringfiles)
`defaultLocale`	((string))	'en'	The default locale for the site. Note that this setting will be overridden for any request that sends an "Accept-Language" header (i.e. most browsers), but it's still useful if you need to localize the response for requests made by non-browser clients (e.g. cURL).
`updateFiles`	((boolean))	false	Whether to automatically add new keys to locale (translation) files when they are encountered during a request.

sails.config.log

Configuration for the instance of the Sails logger (`sails.log`) used in your Sails app. The options are conventionally specified in the config/log.js configuration file.

Properties

Property	Type	Default	Details
`level`	((string))	`'info'`	Set the level of detail to be shown in your app's log

sails.config.models

Your default project-wide **model settings**. Can also be overridden on a per-model basis by providing a top-level property with the same name in that model definition. For more details, see the conceptual docs on Model Settings. These options are conventionally specified in the config/models.js configuration file.

```
sails.config.models;
```

Properties

Property	Type	Default	Details
`attributes`	((object))	`{}`	The basic pieces of information to store about a model. See Attributes.
`migrate`	((string))	*see Model Settings*	How & whether Sails will attempt to automatically rebuild the tables/collections/etc. in your schema
`connection`	((string))	`"localDiskDb"`	The default database connection any given model will use without a configured override
`autoPK`	((boolean))	`true`	Toggle the automatic definition of a primary key in your model
`autoCreatedAt`	((boolean))	`true`	Toggle the automatic definition of a property createdAt in your model
`autoUpdatedAt`	((boolean))	`true`	Toggle the automatic definition of a property updatedAt in your model
`tableName`	((string))	*identity*	Used to specify database table name for the model
`dynamicFinders`	((boolean))	`true`	Toggle the automatic creation of Dynamic Finders

sails.config.policies

What is this?

Policies are like any other system for authentication control. You can allow or deny access in fine granularity with policies.

Description

Your app's ACL (access control list) is located in **config/policies.js**.

Applying a Policy

To a Specific Action

To apply a policy to a specific action in particular, you should specify it on the right-hand side of that action:

```
{
  ProfileController: {
      edit: 'isLoggedIn'
  }
}
```

To an Entire Controller

To set the default policy mapping for a controller, use the `*` notation:

> **Note:** Default policy mappings do not "cascade" or "trickle down." Specified mappings for the controller's actions will override the default mapping. In this example, `isLoggedIn` is overriding `false`.

```
{
  ProfileController: {
    '*': false,
    edit: 'isLoggedIn'
  }
}
```

Globally

> **Note:** Global policy mappings do not "cascade" or "trickle down" either. Specified mappings, whether they're default controller mappings or for specific actions, will **ALWAYS** override the global mapping. In this example, `isLoggedIn` is overriding `false`.

```
{

  // Anything you don't see here (the unmapped stuff) is publicly accessible
  '*': true,

  ProfileController: {
    '*': false,
    edit: 'isLoggedIn'
  }
}
```

Built-in policies

true

> This is the default policy mapped to all controllers and actions in a new project. In production, it's good practice to set this to `false` to prevent access to any logic you might have inadvertently exposed.

Allow public access to the mapped controller/action. This will allow any request to get through, no matter what.

```
module.exports = {
  UserController: {

    // login should always be accessible
    login: true

  }
}
```

false

NO access to the mapped controller/action. No requests get through. Period.

```
module.exports = {
  MathController: {

    // This fancy algorithm we're working on isn't done yet
    // so we set it to false to disable it
    someFancyAlgorithm: false

  }
}
```

Custom policies

You can apply one or more policies to a given controller or action. Any file in your `/policies` folder (e.g. `authenticated.js`) is referable in your ACL (`config/policies.js`) by its filename minus the extension, (e.g. `'authenticated'`).

```
module.exports = {
  FileController: {
    upload: ['isAuthenticated', 'canWrite', 'hasEnoughSpace']
  }
}
```

Multiple Policies

To apply two or more policies to a given action, (order matters!) you can specify an array, each referring to a specific policy.

```
UserController: {
    lock: ['isLoggedIn', 'isAdmin']
}
```

In each of the policies, the next policy in the chain will only be run if `next()` , the third argument, is called. When and if the last policy calls `next()` , the requested controller action is run.

sails.config.routes

Configuration for custom (aka "explicit") routes. `sails.config.routes` consists of a single Javascript object whose keys are URL paths (the "address") and whose values are one of several types of route handler configurations (the "target"), for example:

```
module.exports.routes = {

    "GET /": {view: "homepage"},
    "POST /foo/bar": {controller: "FooController", action: "bar"}

}
```

Please see the routes concept overview for a full discussion of Sails routes, and the custom routes documentation for a detailed description of the available configurations for both the route address and target.

sails.config.session

What is this?

Sails session integration leans heavily on the great work already done by Express, but also unifies Socket.io with the Connect session store.

Description

Sails session integration leans heavily on the great work already done by Express, but also unifies Socket.io with the Connect session store. It uses Connect's cookie parser to normalize configuration differences between Express and Socket.io and hooks into Sails' middleware interpreter to allow you to access and auto-save to `req.session` with Socket.io the same way you would with Express.

`secret`

Session secret is automatically generated when your new app is created. Replace at your own risk in production-- you will invalidate the cookies of your users, forcing them to log in again.

`key`

Session key is set as `sails.sid` by default. This is the name used in the cookie to recover the session.

If you are runing multiple instances of sails, you can lost your session with Websocket. Replace key by an unique name, solve this issue.

Shared Redis session store

In production, uncomment the following line to set up a shared redis session store that can be shared across multiple Sails.js servers.

```
adapter: 'redis',
```

The following values are optional, if no options are set a redis instance running on localhost is expected. Read more about options at: https://github.com/visionmedia/connect-redis

```
        host: 'localhost',
        port: 6379,
        ttl: <redis session TTL in seconds>,
        db: 0,
        pass: <redis auth password>
        prefix: 'sess:'
```

Uncomment the following lines to use your Mongo adapter as a session store

```
        adapter: 'mongo',

        host: 'localhost',
        port: 27017,
        db: 'sails',
        collection: 'sessions',
```

Optional Values:

```
        // Note: url will override other connection settings
        // url: 'mongodb://user:pass@host:port/database/collection',

        username: '',
        password: '',
        auto_reconnect: false,
        ssl: false,
        stringify: true
```

sails.config.sockets

What is this?

These configuration options provide transparent access to Socket.io, the WebSocket/pubsub server encapsulated by Sails.

Commonly-Used Options

Property	Type	Default	Details
`onConnect`	((function))	see config/sockets.js	A function to run every time a new client-side socket connects to the server. This function is **deprecated**. Use `beforeConnect` instead.
`onDisconnect`	((function))	see config/sockets.js	A function to run every time a new client-side socket disconnects from the server. This function is **deprecated**. Use `afterDisconnect` instead.
`adapter`	((string))	`'memory'`	The database where socket.io will store its message queue and answer pubsub logic. Can be set to either `'memory'` or `'redis'`
`host`	((string))	`'127.0.0.1'`	Hostname of your redis instance (only applicable if using the redis socket store adapter)
`port`	((integer))	`6379`	Port of your redis instance (only applicable if using the redis socket store adapter)
`db`	((string))	`'sails'`	The name of the database to use within your redis instance (only applicable if using the redis socket store adapter)
`pass`	((string))	((undefined))	The password for your redis instance (only applicable if using the redis socket store adapter)

Advanced Configuration

These configuration options provide lower-level access to the underlying Socket.io server settings for complete customizability.

Property	Type	Default	Details
`serveClient`	((boolean))	`false`	Whether to serve the default Socket.io client at `/socket.io/socket.io.js`. Occasionally useful for advanced debugging.
`sendResponseHeaders`	((boolean))	`true`	Whether to include response headers in the JWR (JSON WebSocket Response) originated for each socket request (e.g. `io.socket.get()` in the browser) This doesn't affect direct socket.io usage--only if you're communicating with Sails via the request interpreter (e.g. making normal calls with the sails.io.js browser SDK). This can be useful for squeezing out more performance when tuning high-traffic apps, since it reduces total bandwidth usage. However, since Sails v0.10, response headers are trimmed whenever possible, so this option should almost never need to be used, even in extremely high-scale applications.
			A function to run every time a new client-side socket attempts to connect to the server which can be used to reject or allow the incoming connection. Useful for tweaking your production environment to prevent DoS attacks, or reject socket.io connections based on business-specific heuristics (e.g. if stooges from a competing business create bots to post spam links about their commercial product in your public, open-source chat room) (In Sails v0.9 and v0.10, this was called

`beforeConnect`	((boolean)), ((function))	`undefined`	`authorization` -- it has changed due to the upgrade to socket.io v1) To define your own custom logic, specify a function like: `beforeConnect: function (handshake, cb) { /* pass back true to allow, false to deny */ return cb(null, true); }` As of Sails v0.11, Sails no longer blocks incoming socket connections without cookies-- instead, cookies (and by corollary-sessions) are granted automatically. If a requesting socket.io client cannot receive a cookie (i.e. making a cross-origin socket.io connection) the `sails.io.js` socket client will automatically send a CORS+JSONP request to try and obtain one **BEFORE CONNECTING** (refer to the `grant3rdPartyCookie` option above for details). In the antagonistic scenario where even this fails, Sails will still grant a new cookie upon connection, which allows for a one-time session.`
`pingTimeout`	((number))	`60000`	This is a raw configuration option exposed from Engine.io. It reflects how many ms without a pong packet to wait before considering a socket.io connection closed
`pingInterval`	((number))	`25000`	This is a raw configuration option exposed from Engine.io. It reflects the number of miliseconds to wait between "ping packets" (i.e. this is what "heartbeats" has become, more or less)
`maxBufferSize`	((number))	`10E7`	This is a raw configuration option exposed from Engine.io. It reflects the maximum number of bytes or characters in a message when polling before automatically closing the socket (to avoid DoS).

`transports`	((array))	`['polling', 'websocket']`	An array of allowed transport methods which the clients will try to use.
`allowUpgrades`	((boolean))	`true`	This is a raw configuration option exposed from Engine.io. It indicates whether to allow Socket.io clients to upgrade the transport that they are using (e.g. start with polling, then upgrade to a true WebSocket connection).
`cookie`	((string)), ((boolean))	`false`	This is a raw configuration option exposed from Engine.io. It indicates the name of the HTTP cookie that contains the connecting socket.io client's socket id. The cookie will be set when responding to the initial Socket.io "handshake". Alternatively, may be set to `false` to disable the cookie altogether. Note that the `sails.io.js` client does not rely on this cookie, so it is disabled (set to `false`) by default for enhanced security. If you are using socket.io directly and need to re-enable this cookie, keep in mind that the conventional setting is `"io"` .

sails.config.views

Configuration for your app's server-side views. The options are conventionally specified in the `config/views.js` configuration file.

Properties

Property	Type	Default	Details
`layout`	((string)) -or- ((boolean))	`"layout"`	Set the default layout for your app by specifying the relative path to the desired layout file from your views folder (i.e. `views/` .) Or disable layout support altogether with `false` .
`engine`	((string))	`"ejs"`	The view engine your app will use to compile server-side markup into HTML.
`extension`	((string))	Same as `engine`	The file extension for view files.
`locals`	((object))	`{}`	Default data to be included as view locals every time a server-side view is compiled anywhere in this app.

Notes

> - If your app is NOT using `ejs` (the default view engine) Sails will function as if the `layout` option was set to `false` . To take advantage of layouts when using a custom view engine like Jade or Handlebars, check out that view engine's documentation to find the appropriate syntax.

Waterline (ORM)

By default, Sails comes bundled with an ORM called Waterline (implemented in the orm hook.)

Working with Models

This section of the documentation focuses on the model methods provided by Waterline out of the box. In addition to these, additional methods can come from hooks (i.e. the resourceful pubsub methods), be exposed by the underlying adapters to provide custom functionality, or be hand-written in your app to wrap reusable custom code.

> For an in-depth introduction to models in Sails/Waterline, see http://sailsjs.org/documentation/concepts/ORM/Models.html.

```
/**
* Parrot.js
*
* @description :: The set of parrots registered in our app.
* @docs        :: http://sailsjs.org/#!documentation/concepts/Mod
*/

module.exports = {

  attributes: {

    // e.g. "Polly"
    name: {
      type: 'string'
    },

    // e.g. 3.26
    wingspan: {
      type: 'float',
      required: true
    },

    // e.g. "cm"
    wingspanUnits: {
      type: 'string',
      enum: ['cm', 'in', 'm', 'mm'],
      defaultsTo: 'cm'
    },

    // e.g. [{...}, {...}, ...]
    knownDialects: {
      collection: 'Dialect'
```

Built-In Model Methods

In general, model methods are *asynchronous*, meaning you cannot just call them and use the return value. Instead, you must use callbacks, or promises. Most built-in model methods accept a callback as an optional final argument. If the callback is not supplied, a chainable Query object is returned, which has methods like `.where()` and `.exec()`. See Working with Queries for more on that.

Method	Summary
`.create()`	Create record consisting of object passed in
`.find()`	Lookup an array of records which match the specified criteria
`.findOne()`	Lookup a single record which matches the specified criteria, or send back `null` if it doesn't.
`.update()`	Update records matching the specified criteria, setting the specified object of `attrName:value` pairs.
`.destroy()`	Destroy records matching the specified criteria.
`.findOrCreate()`	Lookup a single record which matches the specified criteria, or create it if it doesn't.
`.count()`	Get the total count of records which match the specified criteria.
`.native()` / `query()`	Make a direct call to the underlying database driver.
`.stream()`	Return a readable (object-mode) stream of records which match the specified criteria

`sails.models`

If you need to disable global variables in Sails, you can still use `sails.models.<model_identity>` to access your models.

A model's `identity` is different than its `globalId`. The `globalId` is determined automatically from the name of the model, whereas the `identity` is the all-lowercased version. For instance, you the model defined in `api/models/Kitten.js` has a globalId of `Kitten`, but its identity is `kitten`. For example:

```
// Kitten === sails.models.kitten
sails.models.kitten.find().exec(function (err, allTheKittens) {
  // We also could have just used `Kitten.find().exec(...)`
  // if we'd left the global variable exposed.
});
```

.count([`criteria` ,] `callback`)

Purpose

Returns the number of records in your database that meet the given search criteria.

Overview

Parameters

#	Description	Accepted Data Types	Required ?
1	Find Criteria	`{}` , `[{}]` , `string` , `int`	No
2	Callback	`function`	No

Callback Parameters

#	Description	Possible Data Types
1	Error	`Error`
2	Number of Records	`int`

Example Usage

```
User.count({name:'Flynn'}).exec(function countCB(error, found) {
  console.log('There are ' + found + ' users called "Flynn"');

  // There are 1 users called 'Flynn'
  // Don't forget to handle your errors
});
```

Notes

> Any string arguments passed must be the ID of the record.

.create(`values` , [`callback`])

Purpose

Creates a new instance of this model in the database.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Record(s) to Create	`{}` , `[{}]`	Yes
2	Callback	`function`	No

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`
2	Records Created	`{}` , `[{}]`

Example Usage

```
// create a new record with name 'Walter Jr'

User.create({name:'Walter Jr'}).exec(function createCB(err, created){
  console.log('Created user with name ' + created.name);
});

// Created user with name Walter Jr
// Don't forget to handle your errors and abide by the rules you defined in your model
```

.destroy(`criteria` , [`callback`])

Purpose

Destroys all records in your database that match the given criteria.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Find Criteria	`{}` , `[{}]` , `string` , `int`	Yes
2	Callback	`function`	No

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`
2	Deleted Records	`[{}]`

Example Usage

```
User.destroy({name:'Flynn'}).exec(function deleteCB(err){
  console.log('The record has been deleted');
});

// If the record existed, then it has been deleted
// Don't forget to handle your errors
```

Notes

> If you want to confirm the record exists before you delete it, you must first perform a find() Any string arguments passed must be the ID of the record.

.find(`criteria` , [`callback`])

Purpose

Finds and returns all records that meet the criteria object(s) that you pass it.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Find Criteria	`{}` , `[{}]` , `string` , `int`	Yes
2	Callback	`function`	Yes

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`
2	Found Records	`[{}]`

Example Usage

```
User.find({}).exec(function findCB(err, found){
  while (found.length)
    console.log('Found User with name ' + found.pop().name)
});

// Found User with name Flynn
// Found User with name Jessie

// Don't forget to handle your errors
```

Notes

> Any string arguments passed must be the ID of the record. This method will ALWAYS return records in an array. If you are trying to find an attribute that is an array, you must wrap it in an additional set of brackets otherwise Waterline will think you want to perform an inQuery.

.findOne(`criteria` , [`callback`])

Purpose

This finds and returns a single record that meets the criteria.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Find Criteria	`{}` , `string`	Yes
2	Callback	`function`	Yes

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`
2	Found Record	`{}`

Example Usage

```
User.findOne({name:'Jessie'}).exec(function findOneCB(err, found){
  console.log('We found '+found.name);
});

// We found Jessie
// Don't forget to handle your errors
```

Notes

> Any string arguments passed must be the ID of the record. If you are trying to find an attribute that is an array, you must wrap it in an additional set of brackets otherwise Waterline will think you want to perform an inQuery.
>
> If no matching record is found, the value of `found` will be `undefined`. Not finding a record does *not* constitute an error for `findOne`.

.findOrCreate(`criteria` , `record` , [`callback`])

Purpose

Checks for the existence of the record in the first parameter. If it can't be found, the record in the second parameter is created.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Find Criteria	`{}` , `[{}]` , `string` , `int`	Yes
2	Records to Create	`{}` , `[{}]`	Yes
3	Callback	`function`	No

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`
2	Records Created	`{}` , `[{}]`

Example Usage

```
User.findOrCreate({name:'Walter'}, {name:'Jessie'}).exec(function createFindCB(err, recor
  console.log('What\'s cookin\' '+record.name+'?');
});

// What's cookin' Jessie?
// Don't forget to handle your errors and abide by the rules you defined in your model
```

Notes

> Any string arguments passed must be the ID of the record. If you are trying to find an attribute that is an array, you must wrap it in an additional set of brackets otherwise Waterline will think you want to perform an inQuery.

.native()

`.native()` is only available when using Sails/Waterline with MongoDB.

Returns a raw Mongo collection instance representing the specified model, allowing you to perform raw Mongo queries.

For full documentation and usage examples, check out the native Node Mongo driver.

Note that `sails-mongo` maintains a single Mongo connection for each of your configured connections/datastores. Consequently, when using `.native()`, you don't need to close or open `db` manually. For lower-level usage, you can `require('mongodb')` directly.

Example

```
Pet.native(function(err, collection) {
  if (err) return res.serverError(err);

  collection.find({}, {
    name: true
  }).toArray(function (err, results) {
    if (err) return res.serverError(err);
    return res.ok(results);
  });
});
```

Source: https://gist.github.com/mikermcneil/483987369d54512b6104

Notes

> - This method only works with Mongo! For raw functionality in SQL databases, use `.query()`.

.query()

`.query()` is only available on Sails/Waterline models using a SQL database (PostgreSQL and mySQL) adapter. Its purpose is to perform raw SQL queries.

Example

```
Pet.query('SELECT pet.name FROM pet', function(err, results) {
  if (err) return res.serverError(err);
  return res.ok(results.rows);
});
```

Notes

> This method only works with PostgreSQL and mySQL! use .native() for Mongo.

.stream(`criteria`)

Purpose

This method uses a node write stream to pipe model data as it is retrieved without first having to buffer all of the results to memory.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Find Criteria	`{}` , `[{}]` , `string` , `int`	Yes
2	Custom Write/End Methods	`{}`	No

Returned

	Description	Possible Data Types
1	Error	`Error`
2	Stream of Records	`stream`

Example Usage

UsersController.js

```
module.exports = {

  testStream: function(req, res){

    if (req.param('startStream') && req.isSocket){

        var getSocket = req.socket;

        // Start the stream.  Pipe it to sockets.
        User.stream({name:'Walter'}).pipe(getSocket.emit);

    } else {

      res.view();

    }


  }
}
`
```

views/users/testSocket.ejs

```
<script type="text/javascript">
window.onload = function startListening(){
    socket.on('gotUser',function(data){
      console.log(data.name + ' number ' + data.id + ' has joined the party');
    });
};

</script>
<div class="addButton" onClick="socket.get('/users/testStream/', {startStream:true})">Str
```

Notes

> This method is useful for piping data from VERY large models straight to res. You can also pipe it other places. See the node stream docs for more info. Only the mongo, mysql, and posgresql adapters support this method. This won't work with the disk adapter. Any string arguments passed must be the ID of the record.

.update()

Purpose

Updates existing records in the database that match the specified criteria.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Find Criteria	`{}` , `[{}]` , `string` , `int`	Yes
2	Updated Records	`{}` , `[{}]`	Yes
3	Callback	`function`	No

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`
2	Sucessfully Updated Records	`[{}]`

Example Usage

```
User.update({name:'Walter Jr'},{name:'Flynn'}).exec(function afterwards(err, updated){

  if (err) {
    // handle error here- e.g. `res.serverError(err);`
    return;
  }

  console.log('Updated user to have name ' + updated[0].name);
});
```

Notes

> - An array of primary key values passed to `.update()` for a `collection` association will set the association to contain **only** the records with those primary key values provided. That is- it **unlinks all other** records from the association.
> - Although you may pass .update() an object or an array of objects, it will always return an array of objects.
> - If you specify a primary key (e.g. `7` or `"50c9b254b07e040200000028"`) instead of a criteria object, any `.where()` filters will be ignored.
> - Currently, calling `.populate()` on an `.update()` query has no effect. To populate attributes on the results, you should follow up your update with a `find().populate()` query.

Populated Values

In addition to basic attribute data like email addresses, phone numbers, and birthdates, Waterline can dynamically store and retrieve linked sets of records using associations. When `.populate()` is called on a query, each of the resulting records will contain one or more **populated values**. Each one of those **populated values** is a snapshot of the record(s) linked to that particular association at the time of the query.

The type of a populated value is either:

- `null`, or a plain old JavaScript object (POJO), or *(if it corresponds to a "model" association)*
- an empty array, or an array of plain old JavaScript objects *(if it corresponds to a "collection" association)*

For example, assuming we're dealing with orders of adorable wolf puppies:

```
Order.find()
.populate('buyers')  // a "collection" association
.populate('seller')  // a "model" association
.exec(function (err, orders){

  // this array is a snapshot of the Customers who are associated with the first Order as
  orders[0].buyers;
  // => [ {id: 1, name: 'Rob Stark'}, {id: 6, name: 'Arya Stark'} ]

  // this object is a snapshot of the Company that is associated with the first Order as
  orders[0].seller;
  // => { id: 42941, corporateName: 'WolvesRUs Inc.' }

  // this array is empty because the second Order doesn't have any "buyers"
  orders[1].buyers;
  // => []

  // this is `null` because there is no "seller" associated with the second Order
  orders[1].seller;
  // => null
});
```

Modifying populated values

Changes to populated values are persisted (i.e. saved to the database) by calling `.save()` on the record they are attached to. You cannot call `.save()` directly on a populated value.

Changing or remove the linked record of a "model" association can be accomplished by simply setting the property directly on the original record:

```
orders[1].seller = { corporateName: 'Wolf Orphanage' };
```

"collection" associations, on the other hand, *do* have a couple of special (non-enumerable) methods for associating and disassociating linked records. However, `.save()` must still be called on the original record in order for changes to be persisted to the database.

```
orders[1].buyers.add({ name: 'Jon Snow' });
orders[1].save(function (err) { ... });
```

Example

Finally, to put it all together:

```
Order.find()
.populate('buyers')
.exec(function (err, orders){

  orders[1].buyers.add({ name: 'Jon Snow' });
  orders[1].seller = { corporateName: 'Wolf Orphanage' };
  orders[1].save(function (err) {
    // We successfully created a new Customer named Jon and added
    // him to `order[1]` as one of its "buyers".
    // We also created a new company and set it as `order[1]`'s "seller".
    //
    // If we had provided only a primary key value instead of an object,
    // in both cases Waterline would have tried to associate existing
    // Customer and Company records rather than creating new ones.
  });

});
```

* .add(`primary key`)

Purpose

Used to add records to the join table that is automatically generated during a Many-to-Many association. It accepts either the primary key of the model instance (defaults to record ID) or a new record (object) that you want created and to be associated with.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Records	`{}`, `string`, `int`	Yes

Example Usage

```
User.find({name:'Mike'}).populate('pets').exec(function(e,r){
  r[0].pets.add(7);
  r[0].save(function(err,res){
    console.log(res);
  }
});

/*

{ pets:
   [ { name: 'Pinkie Pie',
       color: 'pink',
       id: 7,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Rainbow Dash',
       color: 'blue',
       id: 8,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Applejack',
       color: 'orange',
       id: 9,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
  name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 19:30:54 GMT-0600 (CST),
  id: 7 }

*/
```

Notes

> - .add() does not accept arrays of any kind. Don't try it.
> - Any string arguments passed must be the primary key of the record.
> - `.add()` alone won't actually persist the change in associations to the databse. You should call `.save()` after using `.add()` or `.remove()`.
> - Attempting to add an association that already exists will throw an error. See here for an example.

*.remove(`primary key`)

Purpose

Used to remove records from the join table that is automatically generated during a many-to-many association. Unlike .add(), it only accepts the primary key of the model instance (defaults to record ID).

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Primary Key	`string` , `int`	Yes

Example Usage

```
User.find({name:'Mike'}).populate('pets').exec(function(e,r){
  r[0].pets.remove(7);
  r[0].save(console.log)
});

  /*

{ pets:
   [ { name: 'Rainbow Dash',
       color: 'blue',
       id: 8,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Applejack',
       color: 'orange',
       id: 9,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
  name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 19:30:54 GMT-0600 (CST),
  id: 7 }

  */
```

Notes

> - Any string arguments passed must be the primary key of the record.
> - `.remove()` alone won't actually persist the change in associations to the databse. You should call `.save()` after using `.add()` or `.remove()`.

Working with Queries

Chainable deferred objects returned from Waterline model methods like `.find()` and `.create()`.

```
var query = Stuff.find();
```

You have likely already interacted with query objects in your Sails app. Most of the time, you probably won't think about them as objects *per se*, rather just another part of the syntax for communicating with the database.

The primary purpose of Waterline query instances is to provide a convenient, chainable syntax for working with your models. Methods like `.populate()`, `.where()`, and `.sort()` allow you to refine database calls *before* they're sent down the wire. When you're ready to fire the query off to the database, you can just call `.exec()`.

Promises

In addition to the `.exec()` method, Waterline queries implement a partial integration with the Bluebird promise library, exposing `.then()` and `.catch()` methods.

```
Stuff.find()
.then(function (allTheStuff) {...})
.catch(function (err) {...});
```

If you are a fan of promises, and have a reasonable amount of experience with them, you should have no problem working with this interface. However if you are not very familiar with promises, or don't care one way or another, you will probably have an easier time working with `.exec()`, which uses standard Node.js callback conventions.

```
Stuff.find()
.exec(function (err, allTheStuff) {...})
```

Query Execution

When you **execute** a query, a lot happens:

```
Zookeeper.find().exec(function (err, zookeepers){
  // would you look at all those zookeepers?
});
```

First, it is "shaken out" by Waterline core into a normalized criteria object. Then it passes through the relevant Waterline adapter(s) for translation to the raw query syntax of your database(s) (e.g. Redis or Mongo commands, various SQL dialects, etc.) Finally, each involved adapter uses its native Node.js database driver to send the query out over the network to the corresponding physical database.

When the adapter receives a response, it is marshalled to the Waterline interface spec and passed back up to Waterine core, where it is integrated with any other raw adapter responses into a coherent result set. At that point, it undergoes one last normalization before being passed back to your callback for consumption by your app.

Notes

> - Waterline model methods will **NOT** return a query instance if an optional callback is directly passed as the final argument. Instead, that callback will be triggered when the query is complete.

.exec(`callback`)

Purpose

This is run at the end of a chain of stringable methods. It signals the adapter to run the query.

Parameters

	Description	Accepted Data Types	Required ?
1	Callback	`function`	Yes

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`
2	Data Returned	`{}` , `[{}]` , `int`

Example Usage

```
// refer to any of the examples above
```

Notes

> The .find() method returns a chainable object if you don't supply a callback. This method can be chained to .find() to further filter your results.
>
> If you don't run .exec(), your query will not execute.

.limit(`integer`)

Purpose

Parameters

	Description	Accepted Data Types	Required ?
1	Number to Return	`int`	Yes

Example Usage

```
var myQuery = User.find();
myQuery.limit(12);

myQuery.exec(function callBack(err,results){
    console.log(results)
    });
```

Notes

> The .find() method returns a chainable object if you don't supply a callback. This method can be chained to .find() to further filter your results.

.populate(`foreignKey` , [`query`])

Purpose

This chainable method is used between .find()/.update() and .exec() in order to retrieve records associated with the model being queried. You must supply the Foreign Key specified in your model config.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Foreign Key	`string`	Yes
2	Query	`object`	No

Example Usage

```
User.find({name:'Mike'}).exec(function(e,r){
  console.log(r[0].toJSON())
})

/*
{ name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
*/

User.find({name:'Mike'}).populate('pets').exec(function(e,r){
  console.log(r[0].toJSON())
});

/*
{ pets:
   [ { name: 'Pinkie Pie',
       color: 'pink',
       id: 7,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
     { name: 'Rainbow Dash',
```

```
        color: 'blue',
        id: 8,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Applejack',
        color: 'orange',
        id: 9,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
  name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
*/

User.find({name:'Mike'}).populate('pets',{color:'pink'}).exec(function(e,r){
  console.log(r[0].toJSON())
});

/*
{ pets:
   [ { name: 'Pinkie Pie',
       color: 'pink',
       id: 7,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) }],
  name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
*/
```

Notes

> Any string arguments passed must be the primary key of the record.

.populateAll([`query`])

Purpose

This chainable method is used between .find()/.update() and .exec() in order to retrieve records associated with the model being queried. All known associations of your model will be populated and the query will be applied to each of them.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Query	`object`	No

Example Usage

```
User.find({name:'Mike'}).exec(function(e,r){
  console.log(r[0].toJSON())
})

/*
{ name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
*/

User.find({name:'Mike'}).populateAll().exec(function(e,r){
  console.log(r[0].toJSON())
});

/*
{ poneys:
   [ { name: 'Twinky',
       color: 'brown',
       id: 1,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
  pets:
   [ { name: 'Pinkie Pie',
```

```
        color: 'pink',
        id: 7,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Rainbow Dash',
        color: 'blue',
        id: 8,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) },
      { name: 'Applejack',
        color: 'orange',
        id: 9,
        createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
        updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) } ],
  name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
*/

User.find({name:'Mike'}).populateAll({color:'pink'}).exec(function(e,r){
  console.log(r[0].toJSON())
});

/*
{ pets:
   [ { name: 'Pinkie Pie',
       color: 'pink',
       id: 7,
       createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
       updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST) }],
  name: 'Mike',
  age: 16,
  createdAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  updatedAt: Wed Feb 12 2014 18:06:50 GMT-0600 (CST),
  id: 7 }
*/
```

Notes

> Any string arguments passed must be the primary key of the record.

.skip(`integer`)

Purpose

Parameters

	Description	Accepted Data Types	Required ?
1	Number to Skip	`int`	Yes

Example Usage

```
var myQuery = User.find();
myQuery.skip(12);

myQuery.exec(function callBack(err,results){
    console.log(results)
    });
```

Notes

> The .find() method returns a chainable object if you don't supply a callback. This method can be chained to .find() to further filter your results.

.sort(`string`)

Purpose

Parameters

	Description	Accepted Data Types	Required ?
1	Sort String	`string`	Yes

Example Usage

```
var myQuery = User.find();

var sortString= 'name ASC';

// Sort strings look like this

// '<Model Attribute> <sort type>'

myQuery.sort('name ASC');

myQuery.exec(function callBack(err,results){
    console.log(results)
    });
```

Notes

> The .find() method returns a chainable object if you don't supply a callback. This method can be chained to .find() to further filter your results.
>
> Other Sort Types include
>
> - ASC
> - DESC

.where(`criteria`)

Purpose

Parameters

	Description	Accepted Data Types	Required ?
1	Criteria Object	`{}`	Yes

Example Usage

```
var myQuery = User.find();
myQuery.where({'name':{startsWith:'W'}});

myQuery.exec(function callBack(err,results){
    console.log(results)
    });
```

Notes

> The .find() method returns a chainable object if you don't supply a callback. This method can be chained to .find() to further filter your results.

Records

A record is a uniquely identifiable object that corresponds 1-to-1 with a database entry; e.g. a row in Oracle/MSSQL/PostgreSQL/MySQL, a document in MongoDB, or a hash in Redis.

```
Order.findOne().exec(function (err, order){
  var record = order;
});
```

For the most part, records are just plain old JavaScript objects (aka POJOs). However they do have a few protected (non-enumerable) methods for formatting their wrapped data, as well as a special method (`.save()`) for persisting programmatic changes to the database.

* .save(`callback`)

Purpose

The `save()` method updates your record in the database using the current attributes. It then returns the newly saved object in the callback.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Callback	`function`	Yes

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`
2	Saved Record	`{ }`

Example Usage

```
User.find().exec(
  function(err,myRecords){

    // Grab a record off the top of the returned array and save a new attribute to it
    var getOneRecord = myRecords.pop();
    getOneRecord.name = 'Hank';
    getOneRecord.save(
      function(err,s){
        console.log('User with ID '+s.id+' now has name '+s.name);
      });
  });

// User with ID 1 now has name Hank

// Don't forget to handle your errors.
// Don't forget to abide by the rules you set in your model
```

Notes

> This is an instance method. Currently, instance methods ARE NOT TRANSACTIONAL. Because of this, it is recommended that you use the equivalent model method instead.

.toJSON()

Purpose

This method also returns a cloned model instance. This one however includes all instance methods. Be sure to read the notes on this one.

Overview

Return Value

	Description	Possible Data Types
	Cloned Record	`{ }`

Example Usage

```
User.find().exec(
  function(err,myRecord){
    var datUser = myRecord.pop().toObject();
    console.log(datUser);
  });

/* { id: 2,
  createdAt: '2013-10-31T22:42:25.459Z',
  updatedAt: '2013-11-01T20:12:55.534Z',
  name: 'Hank',
  phoneNumber: '101-150-1337' } */

User.find().exec(
  function(err,myRecord){
    var datUser = myRecord.pop().toJSON();
    console.log(datUser);
  });

/* { id: 2,
  createdAt: '2013-10-31T22:42:25.459Z',
  updatedAt: '2013-11-01T20:12:55.534Z',
  name: 'Hank' } */



// Don't forget to handle your errors
```

For model

```
module.exports = {
  attributes: {
    name: 'string',
    phoneNumber: 'string',

    // Override the default toJSON method

    toJSON: function() {
      var obj = this.toObject();
      delete obj.phoneNumber;
      return obj;
    }
  }
}
```

Notes

> The real power of toJSON relies on the fact every model instance sent out via res.json is first passed through toJSON. Instead of writing custom code for every controller action that uses a particular model (including the "out of the box" blueprints), you can manipulate outgoing records by simply overriding the default toJSON function in your model.
> You would use this to keep private data like email addresses and passwords from being sent back to every client.
>
> This is an instance method. Currently, instance methods ARE NOT TRANSACTIONAL. Because of this, it is recommended that you use the equivalent model method instead.

.toObject()

Purpose

The toObject method returns a cloned model instance (record) but stripped of all instance methods.

Overview

Return Value

	Description	Possible Data Types
	Cloned Record	`{ }`

Example Usage

See usage in .toJSON()

Notes

> You will only want to use .toObject when overriding the default .toJSON instance method.

* .validate(`callback`)

Purpose

Checks the current keys/values on the record against the validations specified in the attributes object of your model.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Callback	`function`	Yes

Callback Parameters

	Description	Possible Data Types
1	Error	`Error`

Example Usage

```
User.find().exec(
  function(err,myRecords){

    // Grab a record off the top, change it to the wrong data type, then try to validate
    var getOneRecord = myRecords.pop();
    getOneRecord.name = ['Marie','Hank'];
    getOneRecord.name.validate(
      function(err){
        if (err)
          console.log(JSON.stringify(err));
      });
  });

// {"ValidationError":{"name":[{"data":["Marie","Hank"],"message":"Validation error: \"Ma
```

For model

```
module.exports = {

  attributes: {
    name: 'string'

  }

};
```

Notes

> This is shorthand for `Model.validate({ attributes }, cb)` If you `.save()` without first validating, Waterline tries to convert. If it can't, it will throw an error. In this case, it would have converted the array to the string 'Marie,Hank'
>
> There will be no parameters in the callback unless there is an error. No news is good news.
>
> This is an instance method. Currently, instance methods ARE NOT TRANSACTIONAL. Because of this, it is recommended that you use the equivalent model method instead.

WebSockets

Overview

There are two main continents in the world of WebSockets- the client (e.g. browser) and the server (e.g. your routes, controllers, and so forth).

Resourceful PubSub

Overview

For apps that rely heavily on real-time client-server communication--for example, peer-to-peer chat and social networking apps--sending and listening for socket events can quickly become overwhelming. Sails helps smooth away some of this complexity by introducing the concept of Resourceful PubSub (Publish / Subscribe). Every model (AKA *resource*) in your app is automatically equipped with class methods for subscribing sockets to notifications about instance creations, updates and deletions. If you're using the Blueprint API, socket messages are automatically broadcast to subscribed sockets when a model event occurs. If not, you can use the methods described in this section to manually communicate model events to clients.

Listening for events on the client

While you are free to use any Javascript library to listen for socket events on the client, Sails does provide its own Socket Client as a convenient way to communicate with the server. Using the Sails socket client makes listening for resourceful pubsub events as easy as:

```
io.socket.on("<model identity>", listenerFunction)
```

> The *model identity* is typically the lowercased version of the model name, unless it has been manually configured in the model file.

Example

Let's say you have a model named `User` in your app, with a single "name" attribute. First, we'll add a listener for "user" events:

```
io.socket.on("user", function(event){console.log(event);})
```

This will log any notifications about `User` models to the console. However, we won't receive any such messages until we *subscribe* to the existing `User` model instances. If you're using the default blueprints, you can subscribe by making a socket request from the client to `/user`:

```
io.socket.get("/user", function(resData, jwres) {console.log(resData);})
```

This requests the current list of users from the Sails server, and subscribes the client to events about each user. Additionally, if the `autoWatch` setting is on (the default), the client will also be notified whenever a new `User` is created, and will automatically be subscribed to the new user. The callback in this example simply logs the user list to the console. See the socket.get reference for more info about this method.

It's important to note that in order for the subscription to take place, the `/user` request must be made via a websocket call, *not* a regular HTTP request. That is, using an AJAX request (e.g. `jQuery.get("/user")`) will *not* result in the client being subscribed to resourceful pubsub messages about `User`. However, once the subscription is made, *any* changes to models--whether they be the result of a socket call, an AJAX request, even a cURL request from the command line--will cause the client to receive a notification. Continuing with the above example, if you were to open up a new browser window and go to the following URL:

```
/user/create?name=joe
```

You would see something like the following in the console of the first window:

```
{
    data: {
        createdAt: "2014-08-01T05:50:19.855Z"
        id: 1
        name: "joe"
        updatedAt: "2014-08-01T05:50:19.855Z"
    },
    id: 1,
    verb: "created"
}
```

The `verb` indicates the kind of action that occurred. The `id` refers to the instance that the action occurred on, and `data` contains more information about the `User` that was acted upon. Each event type sends back slightly different information; see the individual resourceful pubsub method reference documents for more info.

.message(`models` , `data` , [`request`])

Purpose

Publishes a custom message to a model's subscribers.

	Description	Accepted Data Types	Required ?
1	Record (or ID of record) to send message to	`int` , `string` , `object`	Yes
2	Message payload	`object`	Yes
3	Request	`request object`	No

`message()` emits a socket message using the model identity as the event name. The message is broadcast to all sockets subscribed to the model instance via the `.subscribe` model method.

The socket message is an object with the following properties:

- **id** - the `id` attribute of the model instance
- **verb** - `"messaged"` (a string)
- **data** - the message payload

`data`

Arbitrary data to send to the subscribed sockets.

`request`

If this argument is included then the socket attached to that request will *not* receive the notification.

.publishAdd(`{id}` , `attribute` , `idAdded` , [`request`], [`options`])

Purpose

Publishes a notification when an associated record is added to a model's collection. For example, if a `User` model has an association with the `Pet` model so that a user can have one or more pets available in its `pets` attribute, then any time a new pet is associated with a user `publishAdd` may be called.

	Description	Accepted Data Types	Required ?
1	ID of Updated Record	`int` , `string`	Yes
2	Attribute of associated collection	`string`	Yes
3	ID of associated record that was added	`int` , `string`	Yes
4	Request	`request object`	No
5	Additional Options	`object`	No

`publishAdd()` emits a socket message using the model identity as the event name. The message is broadcast to all sockets subscribed to the model instance via the `.subscribe` model method.

The socket message is an object with the following properties:

- **id** - the `id` attribute of the model instance
- **verb** - `"addedTo"` (a string)
- **attribute** - the name of the model attribute that was added to
- **addedId** - the ID of the record that was added

`request`

If this argument is included then the socket attached to that request will *not* receive the notification.

`options.noReverse`

See the documentation for `publishUpdate` for information about `options.noReverse`. In general, you should not have to set this argument unless you are writing your own implementation of `publishAdd` for a model.

.publishCreate(`data` ,[`request`])

Purpose

PublishCreate doesn't actually create anything. It simply publishes information about the creation of a model instance via websockets. PublishCreate is called automatically by the blueprint `create` action.

	Description	Accepted Data Types	Required ?
1	Data to Send	`object`	Yes
2	Request	`Request object`	No

The default implementation of publishCreate only publishes messages to the firehose, and to sockets subscribed to the model class using the `watch` method. It also subscribes all sockets "watching" the model class to the new instance. The socket message to subscribers will include the following properties:

- **id** - the `id` attribute of the new model instance
- **verb** - `"created"` (a string)
- **data** - an object-- the attributes and values of the new model instance

`data`

An object containing the attributes and values of the new model instance.

`request`

If this argument is included then the socket attached to that request will *not* receive the notification.

Example Usage

UsersController.js

```
module.exports = {

  testSocket: function(req,res){

        var nameSent = req.param('name');

        if (nameSent && req.isSocket){

          User.create({name:nameSent}).exec(function created(err,newGuy){
            User.publishCreate({id:newGuy.id,name:newGuy.name});
            console.log('A new user called '+newGuy.name+' has been created');
          });

        } else if (req.isSocket){

          User.watch(req);
          console.log('User with socket id '+sails.sockets.id(req)+' is now subscribed to

        } else {

          res.view();

        }
    }
}

    // Don't forget to handle your errors
```

views/users/testSocket.ejs

```
<script type="text/javascript">
window.onload = function subscribeAndListen(){
    // When the document loads, send a request to users.testSocket
    // The controller code will subscribe you to the model 'users'
    socket.get('/users/testSocket/');

    // Listen for the event called 'user' emited by the publishCreate() method.
    socket.on('user',function(obj){
      if (obj.verb == 'created') {
         var data = obj.data;
         console.log('User '+data.name+' has been created.');
      }
    });
};

function makeNew(){

    // Send the new users name to the 'testSocket' action on the 'users' controller

    socket.get('/users/testSocket/',{name:'Walter'});
}

</script>
<div class="addButton" onClick="makeNew()">Click Me to add a new 'Walter' ! </div>
```

.publishDestroy(`{id}` , [`request`], [`options`])

Purpose

Publish the destruction of a model

	Description	Accepted Data Types	Required ?
1	ID of Destroyed Record	`int` , `string`	Yes
2	Request	`request object`	No
3	Additional options	`object`	No

`publishDestroy()` emits a socket message using the model identity as the event name. The message is broadcast to all sockets subscribed to the model instance via the `.subscribe` model method.

The socket message is an object with the following properties:

- **id** - the `id` attribute of the model instance
- **verb** - `"destroyed"` (a string)
- **previous** - an object--if present, contains the attributes and values of the object that was destroyed.

`request`

If this argument is included then the socket attached to that request will *not* receive the notification.

`options.previous`

If this is set, it is expected to be a representation of the model before it was destroyed. This may be used to send out additional notifications to associated records.

Example Usage

UsersController.js

```
module.exports = {

  testSocket: function(req,res){

        var nameSent = req.param('name');

        if (nameSent && req.isSocket){

          User.findOne({name:nameSent}).exec(function findIt(err,foundHim){
            User.destroy({id:foundHim.id}).exec(function destroy(err){
              User.publishDestroy(foundHim.id);
            });
          });

        } else if (req.isSocket){

        User.find({}).exec(function(e,listOfUsers){
          User.subscribe(req.socket,listOfUsers);
        console.log('User with socket id '+req.socket.id+' is now subscribed to all of th
        });

        } else {

          res.view();

        }

  }
}

    // Don't forget to handle your errors
```

views/users/testSocket.ejs

```
<script type="text/javascript">
window.onload = function subscribeAndListen(){
    // When the document loads, send a request to users.testSocket
    // The controller code will subscribe you to all of the 'users' model instances (reco
    socket.get('/users/testSocket/');

    // Listen for the event called 'message' emited by the publishDestroy() method.
    socket.on('message',function(obj){
      if (obj.verb == 'destroyed') {
        console.log('User '+obj.previous.name+' has been destroyed .');
      }
    });
};

function destroy(){

    // Send the name to the testSocket action on the 'Users' contoller
    socket.get('/users/testSocket/',{name:'Walter'});
}

</script>
<div class="addButton" onClick="destroy()">Click Me to destroy user 'Walter' ! </div>
```

Notes

> Any string arguments passed must be the ID of the record.

.publishRemove(`{id}` , `attribute` , `idRemoved` , [`request`], [`options`])

Purpose

Publishes a notification when an associated record is removed to a model's collection. For example, if a `User` model has an association with the `Pet` model so that a user can have one or more pets available in its `pets` attribute, then any time a pet is removed from a user's `pets` collection, `publishRemove` may be called.

	Description	Accepted Data Types	Required ?
1	ID of Updated Record	`int` , `string`	Yes
2	Attribute of associated collection	`string`	Yes
3	ID of associated record that was removed	`int` , `string`	Yes
4	Request	`request object`	No
5	Additional Options	`object`	No

`publishRemove()` emits a socket message using the model identity as the event name. The message is broadcast to all sockets subscribed to the model instance via the `.subscribe` model method.

The socket message is an object with the following properties:

- **id** - the `id` attribute of the model instance
- **verb** - `"removedFrom"` (a string)
- **attribute** - the name of the model attribute that was removed from
- **removedId** - the ID of the record that was removed

`request`

If this argument is included then the socket attached to that request will *not* receive the notification.

`options.noReverse`

See the documentation for `publishUpdate` for information about `options.noReverse`. In general, you should not have to set this argument unless you are writing your own implementation of `publishRemove` for a model.

.publishUpdate(`{id}` ,[`changes`], [`request`],[`options`])

Purpose

PublishUpdate updates nothing. It publishes information about the update of a model instance via websockets.

	Description	Accepted Data Types	Required ?
1	ID of Updated Record	`int` , `string`	Yes
2	Updated values	`{}`	No
3	Request	`request object`	No
4	Additional Options	`object`	No

`publishUpdate()` emits a socket message using the model identity as the event name. The message is broadcast to all sockets subscribed to the model instance via the `.subscribe` model method.

The socket message is an object with the following properties:

- **id** - the `id` attribute of the model instance
- **verb** - `"updated"` (a string)
- **data** - an object-- the attributes that were updated
- **previous** - an object--if present, the previous values of the updated attributes

`changes`

This should be an object containing any changed attributes and their new values.

`request`

If this argument is included then the socket attached to that request will *not* receive the notification.

`options.previous`

If the `options` object contains a `previous` property, it is expected to be a representation of the model instance's attributes *before* they were updated. This may be used to determine whether or not to publish additional messages (see the `options.noReverse` flag below for

more info).

`options.noReverse`

The default implementation of `publishUpdate` will, if `options.previous` is present, check whether any associated records were affected by the update, and possibly send out additional notifications. For example, if a `Pet` model has an `owner` attribute that is associated with the `User` model so that a user may own several pets, and the data sent with the call to `publishUpdate` indicates that the value of a pet's `owner` changed, then an additional `publishAdd` or `publishRemove` call may be made. To suppress these notifications, set the `options.noReverse` flag to `true`. In general, you should not have to set this flag unless you are writing your own implementation of `publishUpdate` for a model.

Example Usage

UsersController.js

```
module.exports = {

  testSocket: function(req,res){

        var nameSent = req.param('name');

        if (nameSent && req.isSocket){

          User.update({name:nameSent},{name:'Heisenberg'}).exec(function update(err,updat
            User.publishUpdate(updated[0].id,{ name:updated[0].name });
          });

        } else if (req.isSocket){

        User.find({}).exec(function(e,listOfUsers){
          User.subscribe(req.socket,listOfUsers);
        console.log('User with socket id '+req.socket.id+' is now subscribed to all of th
        });

        } else {

          res.view();

        }
    }
}

    // Don't forget to handle your errors
```

views/users/testSocket.ejs

```
<script type="text/javascript">
window.onload = function subscribeAndListen(){
    // When the document loads, send a request to users.testSocket
    // The controller code will subscribe you to all of the 'users' model instances (reco
    socket.get('/users/testSocket/');

    // Listen for the event called 'user'
    socket.on('user',function(obj){
      if (obj.verb == 'updated') {
        var previous = obj.previous;
        var data = obj.data;
        console.log('User '+previous.name+' has been updated to '+data.name);
      }
    });
};

function doEdit(){

    // Send the name to the testSocket action on the 'Users' contoller

    socket.get('/users/testSocket/',{name:'Walter'});
}

</script>
<div class="addButton" onClick="doEdit()">Click Me to add a new User! </div>
```

.subscribe()

Subscribes the requesting client socket to one or more database records (i.e. model instances).

Usage

```
SomeModel.subscribe(req, ids);
```

-or-

```
SomeModel.subscribe(req, ids, contexts);
```

By default (if no "contexts" are provided), the client socket will receive relevant messages emitted by `.publishUpdate()` , `.publishDestroy()` , `.publishAdd()` and `.publishRemove()` .

> **Important**:
>
> This function does *not actually talk to the database*! In fact, none of the resourceful pubsub methods do. These are just a simplified abstraction layer built on top of the lower-level `sails.sockets` methods, designed to make your app cleaner and easier to debug by using conventional names for events/rooms/namespaces etc.

	Argument	Type	Details
1	`req`	((req))	The request object (`req`). You should only use this method from an action.
2	`ids`	((array))	An array of record ids (primary keys).
3	`contexts`	((array))	An optional array of change-type strings ("contexts"). If provided, the subscribing client socket will only receive messages involving the specified types of changes (e.g. if a "destroy" context is specified, the socket will receive notifications from `publishDestroy()` calls involving this record). Otherwise, if left unspecified, the socket will hear about any published events involving this record.

Note: `subscribe` will only work with requests made over a socket.io connection (e.g. using `io.socket.get()`), *not* over an http connection (e.g. using `jQuery.get()`). See the sails.io.js socket client documentation for information on using client sockets to send WebSockets/Socket.io messages with Sails.

Example

```
subscribeToLouies: function (req, res) {
  if (!req.isSocket) {
    return res.badRequest('Only a client socket can subscribe to Louies.  You, sir, app
  }

  // Let's say our client socket has a problem with people named "louie".

  // First we'll find all users named "louie" (or "louis" even-- we should be thorough)
  User.find({ or: [{name: 'louie'},{name: 'louis'}] }).exec(function(err, usersNamedLou
    if (err) {
      return res.negotiate(err);
    }

    // Now we'll use the ids we found to subscribe our client socket to each of these r
    // "destroy" context.
    User.subscribe(req, _.pluck(usersNamedLouie, 'id'), ['destroy']);

    // Now any time a user named "louie" or "louis" is destroyed, our client socket wil
    // a notification (as long as it stays connected anyways).

    // All done!  We could send down some data, but instead we send an empty response.
    // (although we're ok telling this vengeful client socket when our users get
    //  destroyed, it seems ill-advised to send him our Louies' sensitive user data.
    //  We don't want to help this guy to hunt them down irl.)
    return res.ok();
  });
}
```

Blueprints and .subscribe()

By default, the blueprint `find` and `findOne` actions will call `.subscribe()` to subscribe a requesting socket to all returned records. However, the blueprint `update` and `delete` actions will *not* cause a message to be sent to the requesting socket by default--only to the *other* connected sockets. This is intended to allow the caller of `io.socket.update()` (for example) to use the client-side SDK's callback to handle the server response separately. To force the blueprint actions to send messages to all sockets, *including the requesting socket*, set `sails.config.blueprints.mirror` to `true`.

What is `context`?

If you specify a specific *context* (or array of contexts) to subscribe to, you will only get messages sent in that context. For example, `User.subscribe(socket, user, 'update')` will cause the socket to receive messages only when `publishUpdate` is called for `user`.

Subsequent calls to `subscribe` are cumulative, so if you called `User.subscribe(socket, user, 'destroy')` later with the same socket, that socket would then be subscribed to messages from both `publishUpdate` and `publishDestroy`.

You can omit `context` to subscribe a socket to the default contexts for that model class. The default contexts are defined by the `autosubscribe` property of the model class. If `autosubscribe` is not present, then the default contexts are `update`, `destroy`, `message` (for custom messages), `add:*` and `remove:*` (publishAdd and publishRemove messages for associated models).

.subscribers(`record` ,[`contexts`])

Purpose

Returns an array of sockets that are subscribed to `record`. This can be used in conjunction with lower-level methods like `sails.sockets.emit` to send custom messages to a collection of sockets, or with `.subscribe` to subscribe one group of sockets to a new instance.

	Description	Accepted Data Types	Required ?
1	Record	((object)), ((integer)), ((string))	Yes
2	Contexts to subscribe to	((string)), ((array))	No

Note: `record` can be either an instance of a model, or a model's primary key.

`context`

If you specify a specific context (or array of contexts), you will only get sockets that are subscribed to the specified contexts for the record.

Example Usage

Controller Code

```
// Find user #1
User.findOne(1).exec(function(e,userOne){
    // Get all of the sockets that are subscribed to user #1
    var subscribers = User.subscribers(userOne);
    // Subscribe them all to userOne's best friend, too
    _.each(subscribers, function(subscriber) {
       User.subscribe(subscriber, userOne.bestFriendId);
    });
});
```

.unsubscribe(`request` , `records` , [`contexts`])

Purpose

This method will unsubscribe a socket from one or more model instances.

	Description	Accepted Data Types	Required ?
1	Request	`Request object`	Yes
2	Records	`[]` , `object`	Yes
3	Contexts to unsubscribe from	`string` , `array`	No

Note: `unsubscribe` will only work when the request is made over a socket connection (e.g. using `socket.get`), *not* over an http connection (e.g. using `jquery.get`).

`context`

See `.subscribe()` for a discussion of pubsub contexts. Omit this argument to unsubscribe a socket from all contexts.

Example Usage

Controller Code

```
User.findOne({id: 123}).exec(function(err, userInstance) {
    User.unsubscribe(req.socket, userInstance);
});
```

.unwatch(`request`)

Purpose

This unsubscribes a client from publishCreate events for the model.

	Description	Accepted Data Types	Required ?
1	Request	`request object`	Yes

Note: `unwatch` will only work when the request is made over a socket connection (e.g. using `socket.get`), *not* over an http connection (e.g. using `jquery.get`).

.watch(`request`)

Purpose

This subscribes a client to publishCreate events for the model. Any connections that are "watching" the model class will be automatically subscribed to new model instances that are created using the blueprint `create` method.

	Description	Accepted Data Types	Required ?
1	Request	`request object`	Yes

Note: `watch` will only work when the request is made over a socket connection (e.g. using `socket.get`), *not* over an http connection (e.g. using `jquery.get`).

Blueprints and .watch()

> By default, the blueprint `find` and `findOne` actions will call `.watch()` on the model class. This behavior can be changed for all models by setting the `sails.config.blueprints.autoWatch` to `false`, or for a specific model by setting `autoWatch` to `false` in the model's class file.

Socket Client (`sails.io.js`)

> This section of the docs is about the Sails socket client SDK for the browser. It is written in JavaScript and is also usable on the server.
>
> There are also a handful of community projects implementing Sails/Socket.io clients for native iOS, Android, and Windows Phone.

Overview

The Sails socket client (`sails.io.js`) is a tiny browser library that is bundled by default in new Sails apps. It is a lightweight wrapper that sits on top of the Socket.IO client whose purpose is to make sending and receiving messages from your Sails backend as simple as possible.

The main responsibility of `sails.io.js` is to provide a familiar ajax-like interface for communicating with your Sails app using WebSockets/Socket.io. That basically means providing `.get()` , `.post()` , `.put()` , and `.delete()` methods that let you take advantage of realtime features while still reusing the same backend routes you're using for the rest of your app. In other words, running `io.socket.post('/user')` in your browser will be routed within your Sails app exactly the same as an HTTP POST request to the same route.

Can I use this with...

Yes. The Sails socket client can be used to great effect with any front-end framework-- no matter whether it's angular, backbone, ember, knockout, etc.

Do I have to use this?

No. The Sails socket client is extremely helpful when building realtime/chat features in a browser-based UI, but like the rest of the `assets/` directory, it is probably not particularly useful if you are building a native Android app, or an API with no user interface.

Fortunately, like every other boilerplate file and folder in Sails, the socket client is completely optional. To remove it, just delete `assets/js/depencencies/sails.io.js` .

io.socket.on()

Starts listening for server-sent events from Sails with the specified `eventIdentity`. Will trigger the provided callback function when a matching event is received.

Usage

```
io.socket.on(eventIdentity, function (msg) {
  // ...
});
```

	Argument	Type	Details
1	`eventIdentity`	((string))	The unique identity of a server-sent event, e.g. "recipe"
2	`callback`	((function))	Will be called when the server emits a message to this socket.

Callback

	Argument	Type	Details
1	`msg`	((object))	Message sent from the Sails server

Note that the callback will NEVER trigger until one of your back-end controllers, models, services, etc. sends a message to this socket. Typically that is achieved one of the following ways:

Resourceful Pubsub Methods

- server publishes a message about a record to which this socket is subscribed (see Model.publishUpdate(), Model.publishDestroy(), and Model.subscribe())
- server publishes a message informing all permitted watcher sockets that a new record has been created in the model with the same identity as `eventIdentity` (see Model.publishCreate(http://sailsjs.org/documentation/reference/websockets/resourceful-pubsub/publishCreate.html) and Model.watch())

Low-Level Socket Methods

- server emits a message to all known sockets (see sails.sockets.blast())
- server emits a message directly to this socket (`io.socket`) using its unique id (see sails.sockets.emit())

- server broadcasts to a room in which this socket (`io.socket`) has been allowed to join (remember that a socket only stays subscribed as long as it is connected-- i.e. as long as the browser tab is open)

Example

Listen for new orders and updates to existing orders:

```
io.socket.on('order', function onServerSentEvent (msg) {
  // msg => {...whatever the server published/emitted...}
});
```

Another example, this time using Angular:

> Note that this Angular example assumes the backend calls `publishCreate()` at some point.

```
angular.module('cafeteria').controller('CheckoutCtrl', function ($scope) {

  $scope.orders = $scope.orders || [];

  if (!io.socket.alreadyListeningToOrders) {
    io.socket.alreadyListeningToOrders = true;
    io.socket.on('order', function onServerSentEvent (msg) {

      // Let's see what the server has to say...
      switch(msg.verb) {

        case 'created':
          $scope.orders.push(msg.data); // (add the new order to the DOM)
          $scope.$apply();              // (re-render)
          break;

        default: return; // ignore any unrecognized messages
      }
    });
  }
});
```

Notes

> - When listening for resourceful pubsub calls, the `eventIdentity` is the same as the identity of the calling model (e.g. if you have a model "UserComment", the identity is "usercomment".)
> - For context-- these types of server-sent events are sometimes referred to as "comet") messages.

Handle Socket 'Connect' and 'Disconnect' events

If connection to server was interrupted - server was restarted or some network issue - it is possible to handle these events and subscribe to sockets again.

```
io.socket.on('connect', function(){
    io.socket.get('/messages');
    io.socket.get('/notifications/subscribe/statusUpdates');
});

io.socket.on('disconnect', function(){
    console.log('Lost connection to server');
});
```

socket.delete()

Sends a virtual DELETE request to a Sails server using Socket.io.

Usage

```
io.socket.delete(url, data, function (data, jwres){
  // ...
});
```

	Argument	Type	Details
1	`url`	((string))	The destination URL path, e.g. "/checkout".
2	`data`	((*))	Optional request data- if provided, will be URL encoded and appended to `url` (existing query string params in url will be preserved)
3	`callback`	((function))	Optional callback- if provided, will be called when the server responds.

Callback

	Argument	Type	Details
1	`resData`	((*))	Data received in the response from the Sails server (=== `jwres.body` , equivalent to the HTTP response body.)
2	`jwres`	((JWR))	The JSON WebSocket Response object. Has `headers` , a `body` , and a `statusCode` .

Example

```
<script>
io.socket.delete('/users/9', function (resData) {
  resData; // => {id:9, name: 'Timmy Mendez', occupation: 'psychic'}
});
</script>
```

socket.get()

Sends a virtual GET request to a Sails server using Socket.io.

Usage

```
io.socket.get(url, data, function (data, jwres){
  // ...
});
```

	Argument	Type	Details
1	`url`	((string))	The destination URL path, e.g. "/checkout".
2	`data`	((*))	Optional request data- if provided, will be URL encoded and appended to `url` (existing query string params in url will be preserved)
3	`callback`	((function))	Optional callback- if provided, will be called when the server responds.

Callback

	Argument	Type	Details
1	`resData`	((*))	Data received in the response from the Sails server (=== `jwres.body` , equivalent to the HTTP response body.)
2	`jwres`	((JWR))	The JSON WebSocket Response object. Has `headers` , a `body` , and a `statusCode` .

Example

```
<script>
io.socket.get('/users/9', function (resData) {
  resData; // => {id:9, name: 'Timmy Mendez'}
});
</script>
```

socket.post()

Sends a virtual POST request to a Sails server using Socket.io.

Usage

```
io.socket.post(url, data, function (data, jwres){
  // ...
});
```

	Argument	Type	Details
1	`url`	((string))	The destination URL path, e.g. "/checkout".
2	`data`	((*))	Optional request data- if provided, will be JSON-encoded and included as the virtual HTTP body
3	`callback`	((function))	Optional callback- if provided, will be called when the server responds.

Callback

	Argument	Type	Details
1	`resData`	((*))	Data received in the response from the Sails server (=== `jwres.body` , equivalent to the HTTP response body.)
2	`jwres`	((JWR))	The JSON WebSocket Response object. Has `headers` , a `body` , and a `statusCode` .

Example

```
<script>
io.socket.post('/users', { name: 'Timmy Mendez' }, function (resData) {
  resData; // => {id:9, name: 'Timmy Mendez'}
});
</script>
```

socket.put()

Sends a virtual PUT request to a Sails server using Socket.io.

Usage

```
io.socket.put(url, data, function (data, jwres){
  // ...
});
```

	Argument	Type	Details
1	`url`	((string))	The destination URL path, e.g. "/checkout".
2	`data`	((*))	Optional request data- if provided, will be JSON-encoded and included as the virtual HTTP body
3	`callback`	((function))	Optional callback- if provided, will be called when the server responds.

Callback

	Argument	Type	Details
1	`resData`	((*))	Data received in the response from the Sails server (=== `jwres.body` , equivalent to the HTTP response body.)
2	`jwres`	((JWR))	The JSON WebSocket Response object. Has `headers` , a `body` , and a `statusCode` .

Example

```
<script>
io.socket.put('/users/9', { occupation: 'psychic' }, function (resData) {
  resData; // => {id:9, name: 'Timmy Mendez', occupation: 'psychic'}
});
</script>
```

io.socket.request()

Sends a virtual request to a Sails server using Socket.io.

This function is very similar to `io.socket.get()` , `io.socket.post()` , etc. except that it provides lower-level access to the request headers, parameters, method, and URL of the request.

> This function is provided by the `sails.io.js` JavaScript client, and is accessible in the **browser**.

Usage

```
io.socket.request(options, function (data, jwr)){
  // ...
  // jwr.headers
  // jwr.statusCode
  // jwr.body === data
  // ...
});
```

Example

```
io.socket.request({
  method: 'get',
  url: '/user/3/friends',
  params: {},
  headers: {}
})
```

Sockets (sails.sockets)

Overview

Sails exposes several low-level methods for realtime communication with the client via `sails.sockets` . These methods are implemented using a Socket.io connection which is available as `sails.io` ; however, using the `sails.sockets` methods instead will future-proof your app against possible changes in underlying implementation. If your app is mainly sending messages to the client regarding changes in your models, you should try and use the model PubSub methods instead.

Looking for `sails.io` ?

For raw access to the underlying socket.io singleton, you can still access `sails.io` . But starting with Sails v0.10, you should use `sails.sockets` for most low-level use-cases involving sockets, since `sails.io` may be deprecated in an upcoming release to allow for more flexibility/extensibility in the underlying socket implementation.

sails.sockets.blast()

Broadcast a message to all sockets connected to the server.

```
sails.sockets.blast(data);
```

Or:

- `sails.sockets.blast(eventName, data);`
- `sails.sockets.blast(data, socketToOmit);`
- `sails.sockets.blast(eventName, data, socketToOmit);`

Usage

	Argument	Type	Details
1	eventName	((string))	Optional. Defaults to `'message'` .
2	data	((*))	The data to send in the message.
3	socketToOmit	((Socket))	Optional. If provided, that request socket will **not** receive the message blasted out to everyone else. Useful when the broadcast-worthy event is triggered by a requesting user who doesn't need to hear about it again.

Example

In a controller action...

```
sails.sockets.blast('user_logged_in', {
  msg: 'User #' + req.session.userId + ' just logged in.',
  user: {
    id: req.session.userId,
    username: req.session.username
  }
}, req.socket);
```

Notes

> - The phrase "request socket" here refers to an application-layer WebSocket/Socket.io connection. `req.socket` also exists for HTTP requests, but it refers to the underlying TCP socket at the transport layer, which is different. Be sure and ensure `req.isSocket == true` before using `req.socket` with this method.

sails.sockets.broadcast(`roomName` , [`event`], `data` , [`socketToOmit`])

Broadcast a message to a room.

```
sails.sockets.broadcast(roomName, data);
```

Or:

- `sails.sockets.broadcast(roomName, eventName, data);`
- `sails.sockets.broadcast(roomName, data, socketToOmit);`
- `sails.sockets.broadcast(roomName, eventName, data, socketToOmit);`

Usage

	Argument	Type	Details
1	roomName	((string))	The room to broadcast a message in (see sails.sockets.join)
2	eventName	((string))	Optional. Defaults to `'message'` .
3	data	((object))	The data to send in the message.
4	socketToOmit	((Socket))	Optional. If provided, that socket will *not* receive the message. This is useful if you trigger the broadcast from a client, but don't want that client to receive the message itself (for example, sending a message to everybody else in a chat room).

Example

```
sails.sockets.broadcast('artsAndEntertainment', { msg: 'Hi there!' });
```

Notes

> - The phrase "request socket" here refers to an application-layer WebSocket/Socket.io connection. `req.socket` also exists for HTTP requests, but it refers to the underlying TCP socket at the transport layer, which is different. Be sure and ensure `req.isSocket == true` before using `req.socket` with this method.

sails.sockets.emit(`socketIds` , [`event`], `data`)

Purpose

Send a message to one or more sockets by ID.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	IDs of sockets to receive message	`string` , `array`	Yes
2	Event name	`string`	No
3	Message data	`object`	Yes

Example Usage

```
// Controller action

sayHiToFriend: function(req, res) {
    var friendId = req.param('friendId');
    sails.sockets.emit(friendId, 'privateMessage', {from: req.session.userId, msg: 'Hi!'}
    res.json({
      message: 'Message sent!'
    });
}
```

Notes

> - If the event name is not specified then the "message" event will be used by default. This would allow the target sockets to listen on the "message" event in order to react to the emit.

sails.sockets.id()

Gets the ID of a request socket object.

```
sails.sockets.id(socket);
```

Usage

	Argument	Type	Details
1	socket	((Socket))	A request socket (WebSocket/Socket.io) object e.g. `req.socket` .

Once acquired, the socket object's ID can be used to send direct messages to that socket (see sails.sockets.emit) or get information about the rooms that the socket is subscribed to (see sails.sockets.socketRooms).

Example

```
// Controller action

getSocketID: function(req, res) {
  if (!req.isSocket) return res.badRequest();

  var socketId = sails.sockets.id(req.socket);
  // => "BetX2G-2889Bg22xi-jy"

  return res.ok('My socket ID is: ' + socketId);
}
```

Notes

> - The phrase "request socket" here refers to an application-layer WebSocket/Socket.io connection. `req.socket` also exists for HTTP requests, but it refers to the underlying TCP socket at the transport layer, which is different. Be sure and ensure `req.isSocket == true` before using `req.socket` with this method.

sails.sockets.join()

Subscribes a socket to a generic room.

Usage

```
sails.sockets.join(socket, roomName);
```

	Argument	Type	Details
1	`socket`	((string)) -or- ((socket))	The socket to be subscribed. May be specified by the socket's id or a raw socket object.
2	`roomName`	((string))	The name of the room to which the socket will be subscribed. If the room does not exist yet, it will be created.

Example

In a controller action:

```
subscribeToFunRoom: function(req, res) {
  var roomName = req.param('roomName');
  sails.sockets.join(req.socket, roomName);
  res.json({
    message: 'Subscribed to a fun room called '+roomName+'!'
  });
}
```

Note: `req.socket` is only valid if the action is triggered via a socket request, e.g. `socket.get('/subscribeToFunRoom/someRoomName')`

Notes

> - The phrase "request socket" here refers to an application-layer WebSocket/Socket.io connection. `req.socket` also exists for HTTP requests, but it refers to the underlying TCP socket at the transport layer, which is different. Be sure and ensure `req.isSocket == true` before using `req.socket` with this method.

sails.sockets.leave(`socket` , `roomName`)

Purpose

Unsubscribe a socket from a generic room.

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Socket object	`object`	Yes
2	Room Name	`string`	Yes

Example Usage

```
// Controller action

leaveFunRoom: function(req, res) {
    var roomName = req.param('roomName');
    sails.sockets.leave(req.socket, roomName);
    res.json({
      message: 'Left a fun room called '+roomName+'!'
    });
}
```

Note: `req.socket` *is only valid if the action is triggered via a socket request, e.g.* `socket.get('/leaveFunRoom/someRoomName')`

Notes

> - The phrase "request socket" here refers to an application-layer WebSocket/Socket.io connection. `req.socket` also exists for HTTP requests, but it refers to the underlying TCP socket at the transport layer, which is different. Be sure and ensure `req.isSocket == true` before using `req.socket` with this method.

sails.sockets.rooms()

Purpose

Get the list of all current socket rooms

Overview

Parameters

None.

Example Usage

```
// Controller action

getRoomsList: function(req, res) {
    var roomNames = JSON.stringify(sails.sockets.rooms());
    res.json({
      message: 'A list of all the rooms: '+roomNames
    });
}
```

Note: In Socket.io, all sockets are automatically subscribed to a global room with an empty name (''). This room is not returned as part of the array in `sails.sockets.rooms`

sails.sockets.socketRooms(`socket`)

Purpose

Get the list of rooms a socket is subscribed to

Overview

Parameters

	Description	Accepted Data Types	Required ?
1	Socket	`object`	Yes

Example Usage

```
// Controller action

getMyRooms: function(req, res) {
    var roomNames = JSON.stringify(sails.sockets.socketRooms(req.socket));
    res.json({
      message: 'I am subscribed to: '+roomNames
    });
}
```

Notes

> - The phrase "request socket" here refers to an application-layer WebSocket/Socket.io connection. `req.socket` also exists for HTTP requests, but it refers to the underlying TCP socket at the transport layer, which is different. Be sure and ensure `req.isSocket == true` before using `req.socket` with this method.

sails.sockets.subscribers()

Get the IDs of all sockets subscribed to a room.

```
sails.sockets.subscribers(roomName);
```

Usage

	Argument	Type	Details
1	roomName	((string))	The name of the room whose socket ids should be retrieved. e.g. `'supportchat'`

Example

```
sails.sockets.subscribers('supportchat');
// => ['BetX2G-2889Bg22xi-jy', 'BTA4G-8126Kr32bi-za']
```

Sails.js User Guides

Guides

> This will be a top level documentation section on the sails.js website as soon as we get a little more content for it.

Sails.js was created by the Balderdash team out of necessity in order to quickly and efficiently make rock solid apps for our clients. We then open sourced it so others could do the same. Client work is still what pays the bills at Balderdash and while we love to work on Sails full time, we just can't. This section is part of an ongoing effort to further open up the Sails.js framework to the community and keep Balderdash from constraining it's growth.

How can you help?

Think about how you have used Sails for your project then write a guide about it!

What should I write about?

- If you have a less than common use case, write a guide about it.
- If you had a hard time finding a solution to a particular problem while using Sails, write a guide describing your workaround.
- Are you doing something unconventional with Sails? Write a guide.
- Are you using Sails for your embedded hardware project? Please, for the love of God, write a guide!

Okay, I'll do it! Now what?

Thanks. You're awesome! Now, before you write anything, see the very first user guide in this folder under contributing.md

Legal Disclaimer

Just kidding about the legal disclaimer. Seriously though, thank you for contibuting. If it weren't for the help of folks like you, this project wouldn't be half of what it is today. You guys rock.

Sincerely and Truly

Nick (@uncletammy)

Contributing

Submit Your Own Guide

Submitting your own guide is easy. Read this guide in its entirety before submitting a PR though.

Sails.js Documentation Structure

The documentation on the Sails.js website is automatically pulled down from the `sails-docs` github repo and all of the .md files which contain github flavored markdown are turned in html templates. Every time the `sails-docs` repo changes, the changes are instantly reflected on the website.

In order for this to work, the sails-docs repo must have a particular structure. It's easy to follow though.

The Basics

Every folder must contain a .md file with the same (case sensitive) name. A folder called `SecuRity` must contain `SecuRity.md`. This file will be loaded when someone clicks the link on the guides navigation menu.

Every .md file must contain two `<docmeta>` tags. They are required for automatically generating the navigation. Without these tags, the template is ignored.

The `uniqueID` tag is used by the router on the front-end of the Sails.js website. The value can be anything as it's unique among all the other .md files. We add some random numbers to it just in case.

```
<docmeta name="uniqueID" value="someUniqueName85732">
```

The value of the `displayName` tag determines the link text on the navigation menu. This does not need to be unique.

```
<docmeta name="displayName" value="Name To Appear On Navigation">
```

Making a New Section

Lets say you want to create a guide called `Socket.io Safety` in a new section called `Security`.

- First, create the folder `sails-docs/userguides/security/`
- Next, create the file `sails-docs/userguides/security/security.md`
- Add your introduction/overview of the section to security.md and make sure to include your `<docmeta>` tags
- Now, create `sails-docs/userguides/security/socketio.md` and put the content for your guide in it. Make sure to include `<docmeta>` tags

If there is already an appropriate section for your guide, skip to the last step.

The `Contributed By` section.

This section is entirely optional. Feel free to use it to talk a little about yourself.

Example Stub

Feel free to copy and change the file `guideStub.md`

Contributed By

Nicholas Crumrine

A real West Texas cowboy with an affinity for cats

Link

https://twitter.com/ncrumrine

Organizations

Sails, Balderdash,cluckus

Deployment

Deploying your app

Make sure you see 'before deployment' guide in 'security'

Picking a host

Here are some hosts...

Nodejitsu

Deploying to Nodejitsu

This guide was brought to you by

Name

Bio

Link

Organization

Etc...

Openshift

Deploying to OpenShift

This guide was brought to you by

Name

Bio

Link

Organization

Etc...

Document Stubb

This is a stub!

Here is some information

Contributed By

Name

Bio

Link

Organization

Etc...